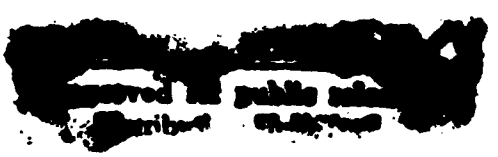
| | |
|---|---|
| **Contractor/Address:** | University of Central Florida<br>Division of Sponsored Research<br>P.O. Box 25000<br>Orlando, FL 32816-0001 |
| **Contract Number:** | N61339-92-C-0016 |
| **Project Title:** | Real-time data filtering and compression in wide area simulation networks |
| **Principal Investigators:** | M. Bassiouni and A. Mukherjee |
| **Telephone No.** | (407) 823-2837 |
| **Title of Report:** | Technical Report<br>CDRL A003 |
| **Date of Report:** | October 2, 1992 |
| **Security Classification:** | Unclassified |
| **Issuing Government Agency:** | DOD/NTSC (STRICOM) |
| **Program Officer/Address:** | Mr. Michael Garnsey<br>STRICOM<br>12350 Research Parkway<br>Orlando, FL 32826-3276 |







# Real-Time Data Filtering and Compression in Wide Area Simulation Networks

## Contract No. N61339-92-C-0016

## Technical Report

This report describes the technical basis of the work performed so far in the area of data filtering and data compression in real-time simulation networks. The report is divided into two parts. Part I presents the results of our effort to design and evaluate data filtering schemes for DIS systems. Detailed algorithms suitable for the implementation of data filtering in the gateways of DIS networks are given. Methods to solve the problem of inaccurate state information at high filtering rates are presented. Part II discusses schemes to enhance the efficiency of Huffman's decoding and similar tree-based codes. A promising scheme, called multibit decoding, is based on the concept of k-bit trees which are used to decode up to k bits at a time. An optimal solution for the mapping of 2-bit trees into memory is presented. The multibit decoding concept offers an attractive way to obtain significant improvement in the speed of Huffman's decoding and is also applicable to other tree-based codes. A detailed description for the encoder/decoder design of a real-time compression chip is given in Appendix I.

# PART I

## Data Filtering in Wide Area Simulation Networks

Achieving the real-time linkage among multiple, geographically-distant, local area networks that support distributed interactive simulation (DIS) is one of the major technical challenges facing the implementation of future large-scale training systems. Data filtering is one of the techniques that can help achieve this real-time linkage [BASS92]. In this report, we present the results of our effort to design and evaluate data filtering schemes for DIS systems. Detailed algorithms suitable for the implementation of data filtering in the gateways of DIS networks are given. Methods to solve the problem of inaccurate state information at high filtering rates are presented.

### Introduction

Today, there is a strong emphasis being placed on the development of efficient "distributed interactive simulation" (DIS) systems [POPE89]. Data filtering and data compression are two complementary techniques that can help improve the networking efficiency of DIS systems. The design of real-time compression for DIS packets will be covered in Part II of this report and in the appendix. In what follows, we concentrate on the technical aspects of designing data filtering algorithms for DIS systems.

Data Filtering refers to the process of analyzing the semantic contents of simulator messages and selecting (for transmission or reception) only the ones that meet certain criterion. For example, if two simulated vehicles, say $V_1$ and $V_2$, are separated by a large distance in the simulated environment, then a state update message from vehicle $V_1$ would be irrelevant to (and would not therefore have to be delivered to) vehicle $V_2$. This example shows the most obvious method of filtering, namely, filtering based on distances in the simulated environment. Other factors (e.g., type

of vehicles) can also affect the filtering process. For example, state update messages from a vehicle submersed in water could normally be ignored by vehicles on the ground. Filtering is used when the total traffic is large enough to overwhelm the small bandwidth of a local site or when the slow nodes in this site cannot handle the fast rate of message arrival. For example, if a high-speed FDDI backbone [ANSI88], [ANSI87], [BASS90] is used to interconnect several 10 Mbits/second Ethernet [IEEE85], [BASS89] simulation networks, filtering could then be used to reduce the size of the traffic flowing from the FDDI backbone to each individual Ethernet LAN. In large scale training exercises, a simulated vehicle would normally need to receive information from only a small subset of the total simulated vehicles at any given time; state update messages from the rest of the vehicles would not be important and can be discarded. The successful implementation of efficient data filtering techniques in network gateways would meet one of the challenges facing the design of long-haul simulation networks.

## An Approach for Data Filtering:

In this section, we shall give the high level details of an approach that can be used for implementing on-the-fly (i.e, real-time) filtering of state update messages. For the purpose of illustrating the basic ideas of the filtering scheme, we shall discuss algorithms relevant to simulators of ground vehicles and we shall use the distance separating these vehicles as the main criterion for filtering.

The filtering scheme uses a one-dimensional vector of distances for each simulated vehicle. The vector is stored in the gateway of the LAN where the simulator resides. Assuming that vehicles in the simulated environment are numbered 1 through n, the vector for the $i^{th}$ simulator will be stored in the form

$$D_i = ( d_{i1} , d_{i2} , \ldots , d_{ii} = 0 , \ldots , d_{in} )$$

where $d_{ij}$ is the distance (in the simulated environment) between vehicle $V_i$ and vehicle $V_j$. For each vehicle, say vehicle $V_i$, we define a *"reachability region"* which specifies a neighborhood region such that the vehicles located within that region are tactically important to vehicle $V_i$ (e.g., they are visible to vehicle $V_i$ or can be affected by it). State update messages from vehicles outside this reachability region need not be delivered to vehicle $V_i$. The reachability region can be simply represented by a reachability radius $R_i$ that gives the maximum distance from vehicle $V_i$ at which another vehicle is reachable (visible). In addition to the distances vector $D_i$, a bit vector $B_i$ is maintained for vehicle $V_i$ and is defined by

$$B_i = ( b_{i1} , b_{i2} , \ldots , b_{ii} = 1 , \ldots , b_{in} )$$

where

$$b_{ij} = 1 \quad \text{if} \quad d_{ij} \leq s\, R_i$$
$$= 0 \quad \text{otherwise}$$

and s is a safety scale factor that suppresses the filtering of messages from vehicles that are outside the reachability region but which are close enough to its border. As shown in Figure 1, a safety ring of depth $(s-1)R_i$ is created to guard against any delay by the filtering mechanism in resuming the delivery of messages sent by a fast vehicle that suddenly entered the reachability region. Thus for example, if s is equal to 1.2, then vehicle $V_i$ will start receiving messages from another vehicle even though that vehicle is at a distance 20% larger than the actual reachability radius. This scheme can be extended such that a different scale factor is used for each vehicle depending on its type and the type of its current surrounding terrain.

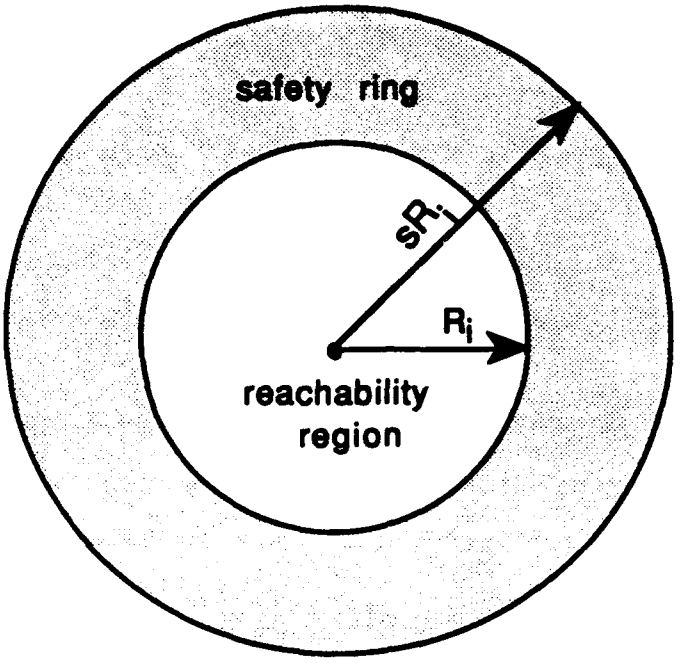



Figure 1. The reachability region

$B_i$ is a binary vector and is therefore more suitable than $D_i$ for real-time filtering decisions. Upon receiving a state update message, say $M_j$ , sent by vehicle $V_j$, the gateway will perform the following algorithm to update the vector $B_i$.

```
Update position of vehicle Vj based on Mj
for i = 1 to n and i ≠ j do
      if bij= 0 and dij ≤ sRi  then bij = 1
      else if bij = 1 and dij > sRi  then bij = 0
      endif
endfor
```

Because of the safety region, the above procedure does not represent a time critical computation; it can in fact be performed as a background job. More details about our approach for the real-time distributed implementation of data filtering will be given shortly. Using the above scheme, the filtering decision becomes an easy task. For example, to determine whether vehicle $V_i$ needs to receive a message $M_j$ sent by vehicle $V_j$, the following code is executed

if $b_{ij} = 1$ then send $M_j$ to vehicle $V_i$
else discard $M_j$

Data filtering is based on the concept of distributed distance computations. Concurrently, the gateway node in each LAN computes the filtering environment for each node in its site. For example, consider the reachability ring and safety region of some static vehicle, say V0, which is surrounded by six moving vehicles V1, V2, ..., and V6. As a result of the movements depicted in Figure 2, the filtering status of vehicles V4, V5, and V6 with respect to V0 will be reversed; thus messages from vehicle V5 will be discarded while those from V4 and V6 will be delivered to V0. On the other hand, the filtering status of vehicles V2, and V3 will not change (messages from V3 will be delivered to V0 while those from V2 will be filtered out). Vehicle V1 will have its status reversed temporarily then will continue to have its message discarded.

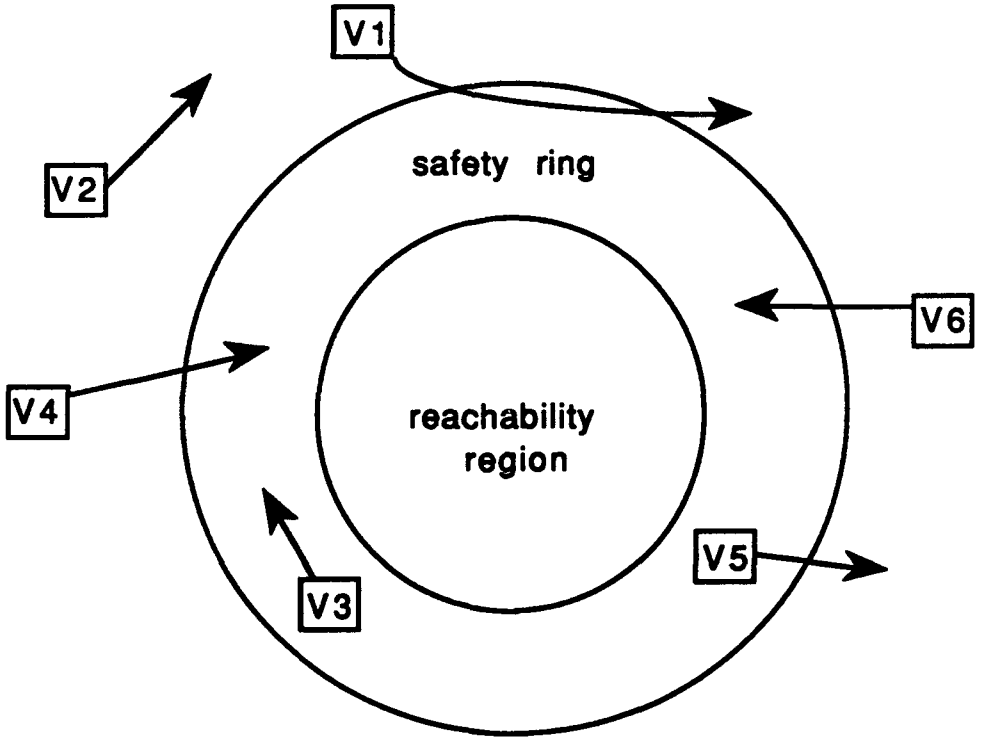



Figure 2. Reachability ring and safety region of vehicle V0

## Implementation of Data Filtering

Filtering should be performed by network gateways at the transmission and reception of a message as well as during its routing in intermediate gateways. Filtering at transmission and routing is the main process that could eliminate the majority of the unneeded messages. Filtering at reception performs a final check and could eliminate the unneeded messages that have not been detected during the transmission and routing phases. For purposes of illustration, we shall discuss the implementation of filtering using the bit vector approach presented in the previous section. Notice that the gateway handles simulator messages in two different ways: 1) the gateway receives messages from nonlocal simulators (called external senders) and distributes them to the simulators on its local site, and 2) the gateway receives messages sent by the local simulators (called local senders) and transmits them over long-haul links to the simulators in other sites. The first case requires *filtering at reception* (i.e., filtering after receiving a message via long-haul links) and the second case requires *filtering at transmission* (i.e., filtering before transmitting a message onto long-haul links). We shall start by discussing filtering at reception then proceed to examine filtering at transmission.

The receiving gateway would need to keep accurate information about the positions of the vehicles simulated by the local nodes connected to it. This can be done without much difficulty since the gateway receives every state update message transmitted by any node in its local site. Without loss of generality, let us assume that the total number of nodes (simulators) in all sites is n, and that the local site under our consideration contains the first m nodes, i.e., its nodes are numbered 1 through m. According to our proposed scheme, the gateway in this site maintains a collection of binary vectors equivalent to a binary matrix, called the filtering matrix B. In the case of filtering at reception, this matrix is defined as

$$B = [\, b_{ij} \,] \quad 1 \le i \le m, \; m+1 \le j \le n$$

where $b_{ij}$ is a filtering flag that is set to 0 if messages from the external simulator j are not relevant (i.e., need not be delivered) to the local

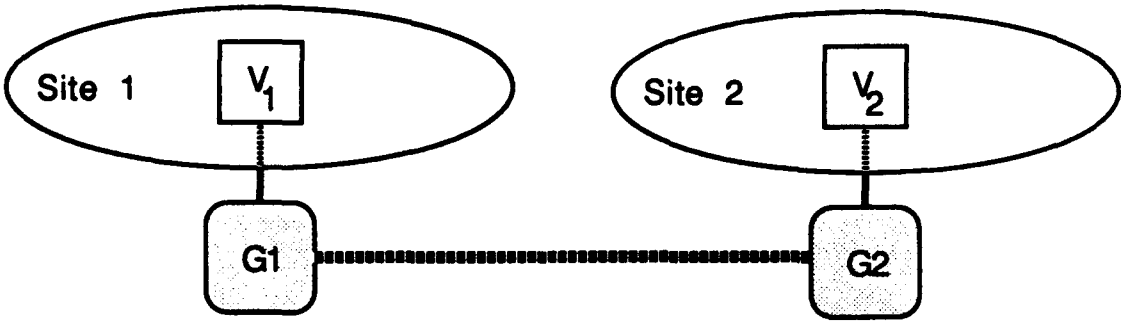
simulator i. As before, the safety scale factor is denoted by s and the reachability region of vehicle $V_i$ is represented by a circle of radius $R_i$. The entire operation of filtering at reception can now be described by the following concurrent code (the PAR construct indicates parallel activities).

```
Algorithm FILTER_AT_RECEPTION;
COBEGIN
loop forever
      Wait for a new message Mj
      Update position of vehicle Vj
      add j to U_LIST /* U_LIST is the update list */
      If j ≤ m then
            /* local sender */
            Call FILTER_AT_TRANSMISSION,
      else  /* external sender */
      begin
            L := Φ  /* empty local list */
            for i=1 to m do
                  if bij =1 then L:=L ∪ {i};
            endfor;
            If L = Φ then discard Mj
            else send  Mj to members of L  endif;
      endif;
endloop;
PAR
/* background update */
loop forever
      wait until U_LIST ≠Φ
      k := First (U_LIST);
      if k ≤ m then
            /* k is local */
            for j=m+1 to n do
                  if dkj ≤ sRk then bkj :=1
                  else bkj :=0 endif;
            endfor;
      else  /* k is external */
            for i=1 to m do
                  if dik ≤ sRi then bik :=1
                  else bik := 0 endif;
            endfor;
      endif;
endloop;
COEND
```

Algorithm FILTER_AT_TRANSMISSION uses a logic similar to that used in the above algorithm; therefore its code will not be given. The main idea can be briefly described as follows. If a local simulator sends a message, the gateway will perform filtering to transmit the message to only those external simulators that can be affected by it (or discard the message if it is not important to any external simulator). There is however a serious problem with this scheme. If the filtering mechanism becomes very successful, the gateways will be deprived of receiving messages from some external simulators. This in turn will make the information (on external vehicles) maintained by each gateway less accurate and can render the filtering decisions incorrect. This problem is discussed next.

### The Problem of Inaccurate State Information

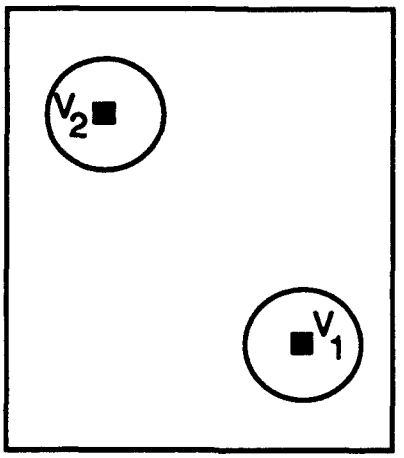
A simple example will be used to illustrate this problem. Consider two vehicle simulators $V_1$ and $V_2$ located in two different DIS sites (LANs). The two sites communicate over long-haul links using the services of the two gateways G1 and G2 as shown in Fig. 3.




Long-haul Network with two DIS sites

Fig. 3

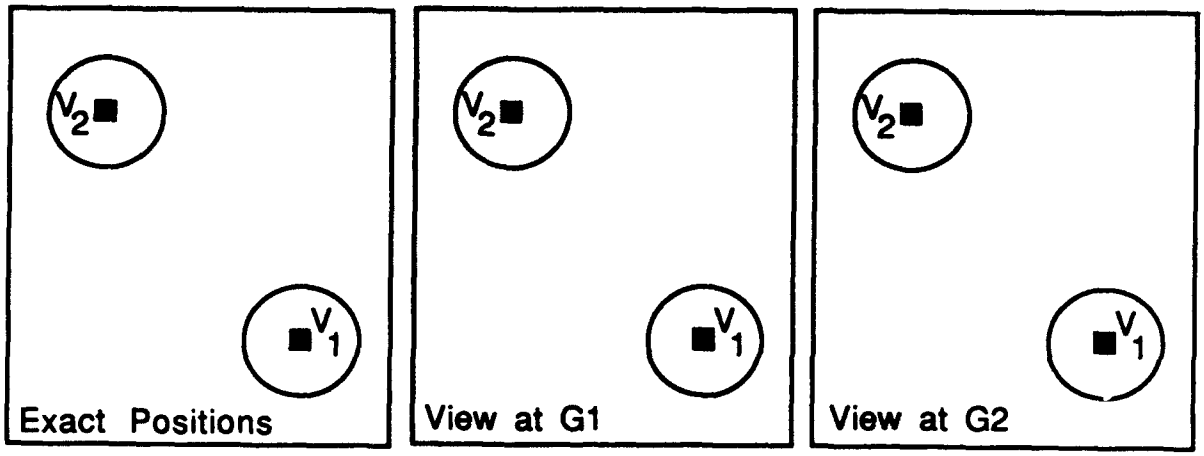
Fig.4 shows the initial positions of the two simulated vehicles in the simulated battlefield. The two vehicles are quite far from each other; each vehicle is outside the reachability region of the other vehicle.



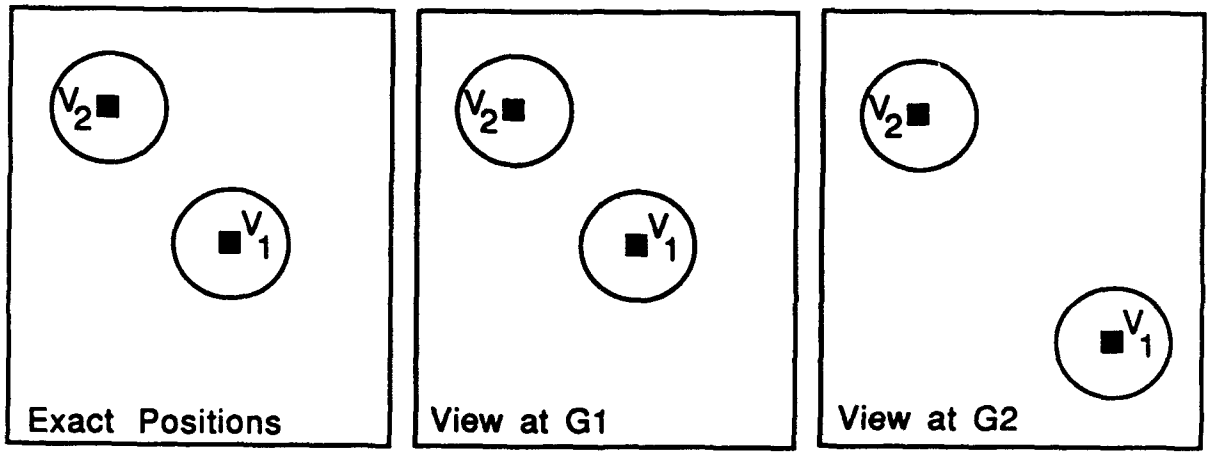

Exact positions of simulated vehicles

Fig. 4

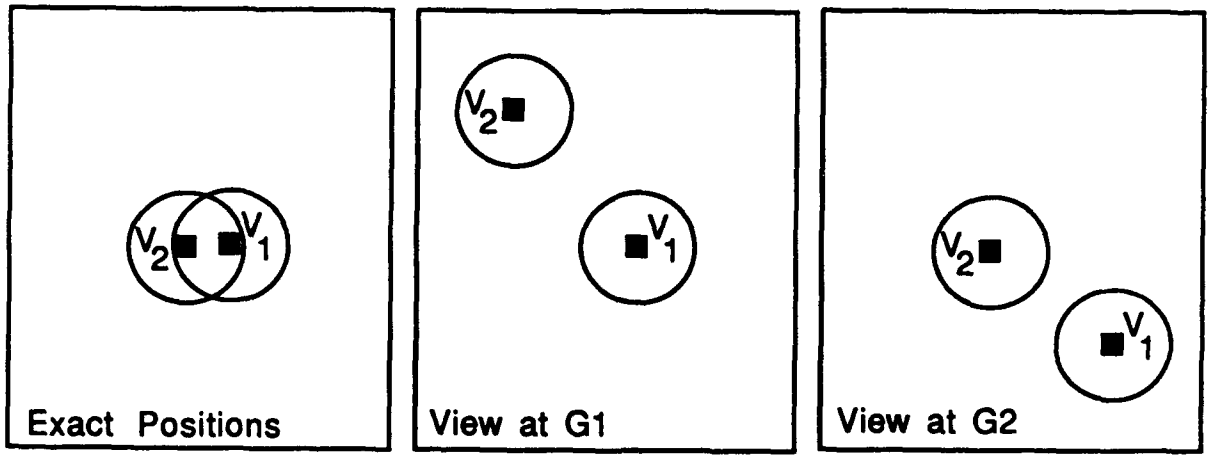
Now assume that vehicle $V_1$ started moving towards vehicle $V_2$. Gateway G1 will execute the Filtering-at-Transmission algorithm and will find that the state update messages emitted by $V_1$ need not be delivered to $V_2$. G1 will therefore refrain from sending these messages to G2. Thus this latter gateway continues to have the initial position of vehicle $V_1$ (i.e. the position shown in Fig 4). Now if vehicle $V_2$ moves towards $V_1$, gateway G2 will determine that the state update messages emitted by $V_2$ need not be delivered to $V_1$. G2 will therefore refrain from sending these messages to G1. The result is that G1 will have inaccurate information about the position of $V_2$. A situation can subsequently arise where the two vehicles $V_1$ and $V_2$ are near each other but each one of them is deprived of receiving the state update messages of the other. Fig. 5 depicts the steps of this scenario.




Initially, all views are identical




Gate G2 does not receive update messages from moving vehicle $V_1$




Gate G1 does not receive update messages from moving vehicle $V_2$

Fig. 5

To solve this problem, we use a **dead-reckoning algorithm** similar to that used by the vehicle simulators themselves. This approach is described next.

## Dead Reckoning in Network Gateways

One of the crucial aspects in DIS local and wide area networks is the ability of each simulator participating in an exercise to represent, accurately and in real-time, the state of other simulated vehicles participating in the same exercise. The concept of dead reckoning is used to reduce the number of state update messages that need to be transmitted by each simulator for the purpose of maintaining accurate state representation. Simply, each simulator has a high fidelity model which maintains accurate information (position, speed, velocity, etc.) about its own state. Each simulator also maintains a less accurate model, called the dead reckoning model, for each simulator (including itself) participating in the exercise. The dead reckoning model of a vehicle is periodically updated by extrapolating the information reported in the last state update message of that vehicle. Using first-order extrapolation, the anticipated position of a simulator is obtained by extrapolating its last reported position based on its last reported velocity as follows:

$$X(t + \tau) = X(t) + V_X(t)\,\tau$$
$$Y(t + \tau) = Y(t) + V_Y(t)\,\tau$$
$$Z(t + \tau) = Z(t) + V_Z(t)\,\tau$$

where $X(t)$, $Y(t)$, $Z(t)$ are the World Coordinates of the simulated vehicle at time t as reported in the last state update message, $V_X(t)$, $V_Y(t)$, $V_Z(t)$ are the x, y, z components of the velocity vector of the vehicle at time t, and $X(t + \tau)$, $Y(t + \tau)$, $Z(t + \tau)$ are the new coordinates predicted at $\tau$ units of time after the last state update message.

The prediction of the dead reckoning algorithm can be generally improved by resorting to higher order extrapolation equations. For example, the dead reckoning equations for position using second-order extrapolation are as follows

$$X(t+\tau) = X(t) + V_X(t)\,\tau + 0.5\,A_X(t)\,\tau^2$$
$$Y(t+\tau) = Y(t) + V_y(t)\,\tau + 0.5\,A_y(t)\,\tau^2$$
$$Z(t+\tau) = Z(t) + V_Z(t)\,\tau + 0.5\,A_Z(t)\,\tau^2$$

where $A_X(t)$, $A_y(t)$, $A_Z(t)$ are the x, y, z components of the acceleration vector at time t. In a similar way, third-order extrapolation equations or even higher derivatives can be used in an attempt to improve the accuracy of predictions in dead-reckoning algorithms.

Whenever a state update message is received from a simulator, the information of that message is used to correct the extrapolated information of the dead reckoning model. Finally, when the state of a simulator actually changes, the simulator updates its own high fidelity model and compares it with the extrapolated information of its own dead reckoning model. If there is a large enough discrepancy between the two models, the simulator transmits a new state update message to all other simulators.

The corresponding dead-reckoning approach in network gateways can now be described as follows:

1) Each gateway will maintain accurate information (position, speed, velocity, etc.) about each of the local simulators in its own site. This information (called the high fidelity model) should be reasonably accurate since the gateway receives every message transmitted by a local node.

2) Each gateway also maintains a less accurate model (called the dead reckoning model) for external simulators. The dead reckoning model is obtained by extrapolating the last reported location of each external vehicle based on its last reported velocity. Whenever a message is actually received from an external simulator, the information of that message is used to correct the extrapolated information of the dead reckoning model.

3) Finally each gateway also keeps a dead reckoning model for its local simulators (using the same extrapolation equations used by other gateways). When the gateway receives a message from a local simulator, it updates its high fidelity model and compares it with the extrapolated information of the dead reckoning model. If there is a large enough discrepancy between the positions of the local vehicle in the two models, the gateway transmits the message over the long-haul links.

**Preliminary Performance Results:**

A simulation program has been written and is currently being used to evaluate the data filtering designs. In this section, we present some preliminary results for a configuration with four different LANs. As our tests proceed, we shall submit more results and analysis in future progress and technical reports. The results reported below correspond to periods of peak activities (i.e., majority of the vehicles are moving). The tests were repeated using different values for the radius of the reachability plus safety region. We define the safety period, T, to be the amount of time needed for a vehicle moving in a straight line with a constant speed (equal to the average velocity of moving vehicles)to travel a distance equal to sR, where R is the radius of the reachability circle and s the safety factor. Fig. 6 plots the relationship between the safety time in hours and the overall filtering rate. The latter is defined to be the average percentage of messages that get filtered out (at transmission or at reception).

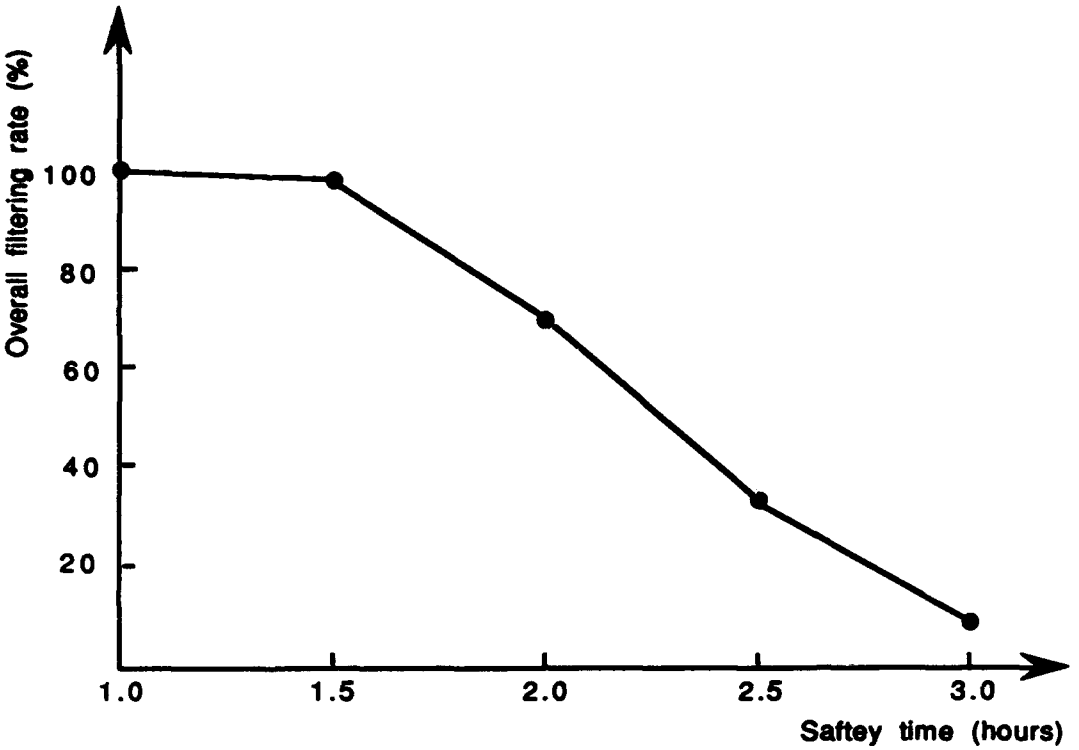



Fig. 6. Filtering rate vs. safety time

Tables 1 through 3 give the detailed results for filtering at transmission (FaT) and filtering at reception (FaR) at selected values of safety time.

Table 1. Filtering at safety time of 1.5 hours

| LAN No. | FaT | FaR |
|---|---|---|
| 1 | 81.6% | 84.6% |
| 2 | 85.9% | 85.2% |
| 3 | 76.8% | 81.4% |
| 4 | 82.4% | 82.9% |

Table 2. Filtering at safety time of 2.0 hours

| LAN No. | FaT | FaR |
|---|---|---|
| 1 | 36.0% | 51.6% |
| 2 | 29.9% | 50.3% |
| 3 | 37.5% | 57.4% |
| 4 | 38.8% | 55.1% |

Table 3. Filtering at safety time of 2.5 hours

| LAN No. | FaT | FaR |
|---|---|---|
| 1 | 21.6% | 26.6% |
| 2 | 9.1% | 21.5% |
| 3 | 16.1% | 26.0% |
| 4 | 9.9% | 17.7% |

## References


[ANSI88] ANSI Standard X3.148 "FDDI token-ring physical layer protocol (PHY)" American National Standards, 1988.

[ANSI87] ANSI Standard X3.139 "FDDI token-ring media access control (MAC)" American National Standard, 1987.

[BASS92] Bassiouni, M.; Mukherjee, A. and Garnsey, M. "Algorithms for message filtering and reduction in real-time networks" Proc. First International Conf. on Computer Communications and Networks, 1992, pp. 62- 66.

[BASS90] Bassiouni, M. and Thompson, J. "Application of FDDI/XTP network protocols to distributed simulation" Proc. of the 12th I/ITS Conference, 1990, pp. 226-233.

[BASS89] Bassiouni, M.; Georgiopoulos, M. and Thompson, J. "Real time simulation networking: network modeling and protocol alternatives" Proceedings of the 11th Interservice/Industry Training Systems Conference (I/ITSC), November 1989, pp. 52-61.

[IEEE85] IEEE/ANSI Standard 8802/3 "Carrier sense multiple access with collision detection (CSMA/CD) access method and physical layer specification" IEEE Computer Society Press, 1985.

[POPE89] Pope, Arthur "The SIMNET network and protocols" BBN Report No. 7102, BBN Communications Corporation, MA, July 1989.

# PART II

# Real-time Data Compression

In this part, we discuss two schemes to enhance the efficiency of Huffman's decoding. The first scheme, called multibit decoding, is based on the concept of k-bit trees which are used to decode up to k bits at a time. An optimal solution for the mapping of 2-bit trees into memory is presented. The multibit decoding concept offers an attractive way to obtain significant improvement in the speed of Huffman's decoding and is also applicable to other tree-based codes. A detailed description for the encoder/decoder design of a real-time compression chip is given in Appendix I. The second scheme, called the multigroup scheme, is suitable for files that exhibit the property of locality of symbol references. The scheme improves the Huffman's compression efficiency as well as the time overhead of the Huffman's decoding process. A multigroup decoding algorithm that works for one-level and two-level hierarchies having arbitrary number of groups is presented. The multigroup technique can be further enhanced by incorporating the multibit concept into its decoding logic.

## Introduction

One of the popular data compression methods is the Huffman's encoding technique [HUFF52] which takes advantage of the skewness of the frequency of input symbols. Accordingly, the most frequent symbols are assigned to the shortest codes and all larger codes are constructed so that shorter codes do not appear as prefixes. Simply, the Huffman's method builds a decode tree (i.e., binary tree in which leaf nodes represent symbols) having minimal external path length. If the set of symbols is given by $\{A_1, A_2, \ldots, A_V\}$, the probability of occurrence of symbol $A_k$ is $p_k$, and the distance from the root of the tree to the leaf node corresponding to symbol $A_k$ is $d_k$, then the Huffman tree minimizes the quantity $\sum_{j=1}^{V} p_j * d_j$. Huffman's compression can be used in scientific databases [BASS85] and is also used in the JPEG image compression standard to store the AC values obtained via DCT coding. Huffman's encoding, arithmetic coding [WITT87] and the LZW scheme [WELC84] are used in

conjunction with lossy schemes to improve the fidelity of compressed images at a given level of compression [BASS91].

Fig. 2.1 gives an example Huffman tree for the twelve symbols A, B, C, D, E, F, G, H, I, J, K, L whose weights (frequencies of occurrence) are assumed to be 4, 3, 4, 1, 1, 2, 8, 2, 1, 1, 1, and 4, respectively. During decoding, the compressed file is processed serially (one bit at a time) and the Huffman tree is repeatedly traversed from its root to the leaf nodes. For example, the bit sequence "001" causes a movement from the root of the Huffman tree of Fig. 2.1 to the leaf node of symbol B. In this section, we concentrate on the problem of improving the efficiency of the decoding (decompression) process of Huffman and other similar compression schemes. Appendxi I covers details of the integrated encoder and decoder design for the Multibit approach.

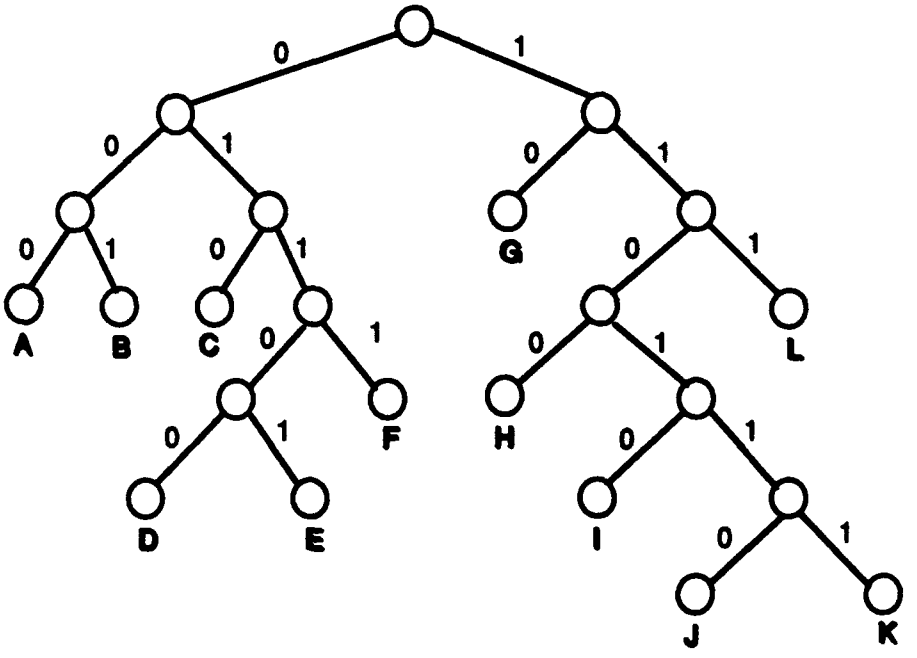



Fig. 2.1. An example Huffman tree

Improving the decoding process is important since the decoding phase in tree-based compression schemes is bit-serial and is therefore inherently slow; bit-serial decoding can benefit the most from better implementations. In the following sections, we shall discuss two schemes for enhancing the efficiency of the Huffman's decoding process. The first scheme, called the multibit (or k-

bit) decoding, is used to reduce the time overhead of the bit-serial decoding operation. The scheme does not require any change in the encoding operation and is applicable to other tree-based codes (a discussion of these codes and their properties is given in [LELE87]). The second scheme, called the multigroup scheme, is useful for data files that exhibit the property of locality of symbol references. For such files, the multigroup scheme improves the Huffman's compression efficiency as well as the time overhead of the decoding process. We shall begin our presentation by discussing the k-bit decoding algorithm then proceed to cover multigroup compression. The terms "k-bit decoding" and "multibit decoding" will be used interchangeably throughout this report.

## K-Bit Decoding

In Huffman's decoding, the compressed bit stream is processed serially one bit at a time. Basically, the decoding operation produces data characters (symbols) by repeatedly traversing the Huffman tree, from the root to the leaf node, under the control of the input bits; a bit value of 1 initiates a visit to the right child while a value of 0 results in a visit to the left child. This process is inherently slow and, because of its strict sequential nature, is not amenable to elegant parallel implementations. The availability of a large number of processors within a parallel machine, for example, may be used to simultaneously decode several files (or records) that were encoded separately, but the sequential decoding of each file needs only one processor at a time and gains no appreciable improvement by the increased scale of parallel hardware. A viable approach to improve the speed, however, can be based on a different concept, namely, using k bits at a time in each step of the decoding process. The problem of "multibit" or "k-bit" decoding has been motivated in [MUKH91a] which also presented a high-level VLSI design for a basic k-bit encoder/decoder. In this report, we give a new formulation for the k-bit decoding problem, present the optimal memory mapping for 2-bit Huffman trees and discuss the basic algorithmic aspects of the k-bit decoding process.

Consider the Huffman tree shown in Fig. 2.1. The first step is to obtain the corresponding k-bit tree. Each edge in this tree corresponds to the encoding of a maximum of k bits of the code. If the length of the code-word is n bits, it is represented by a sequence of $\lceil (n/k) \rceil$ edges in the unique path from the

root to the leaf node; only the last edge leading to the leaf node could possibly have a label with less than k bits. The tree of Fig. 2.1 can be viewed as a 1-bit tree; the corresponding 2-bit tree is shown in Fig. 2.2 (labels inside nodes represent the id or node # of each node). The code of a character in a k-bit tree is obtained by concatenating the labels read from the root to the leaf node of that character.

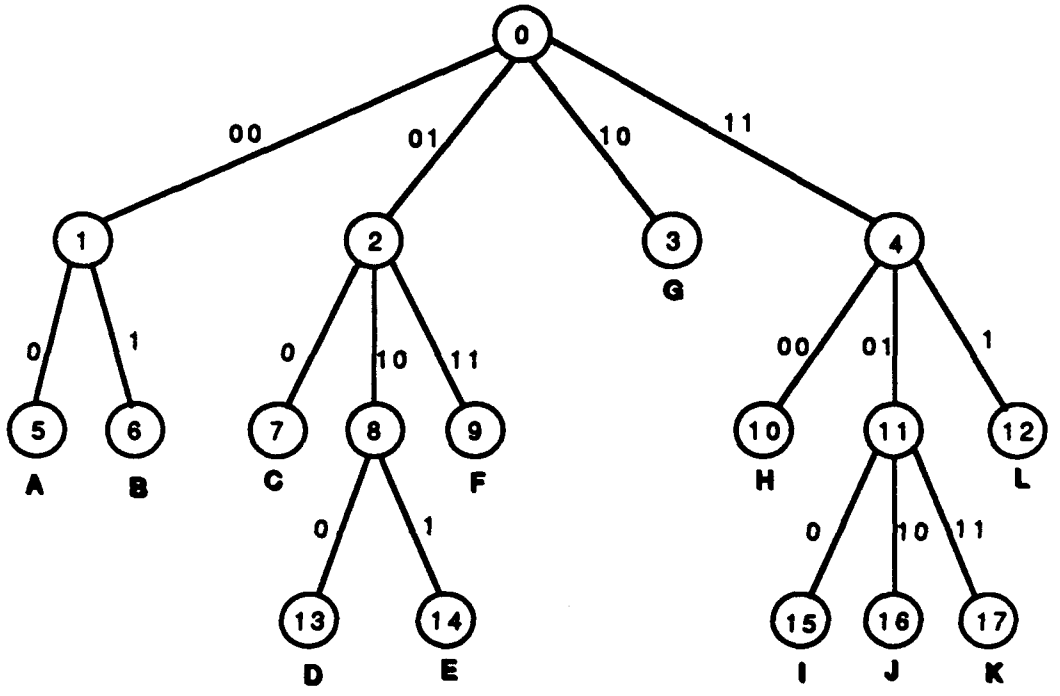



Fig. 2.2. A corresponding 2-bit Huffman tree

**The K-Bit Decode Table**

The purpose of the k-bit scheme is to achieve faster decoding by processing k bits at a time. To maximize the benefit obtained by this scheme, the overhead associated with processing the k-bit sequences should be minimized. In particular, the k-bit decode table must be carefully designed to allow for fast lookup and tree traversal. Below, we discuss a design approach that, in addition to being suitable for efficient software implementation, is quite attractive and suitable for VLSI and associative memory technology.

Consider a k-bit Huffman tree whose nodes are numbered 0, 1, ..., N (assume 0 is the index of the root as shown in Fig. 2.2). A table of size $M \geq N$ is used to store

appropriate information about the N non-root nodes of the k-bit tree (we call this table the k-bit decode table). The corresponding information for the root of the tree will be stored in a separate global record to speed up its access. A non-root node j in this tree, $1 \leq j \leq N$, is mapped to a unique entry r in the table, $0 \leq r \leq M-1$. The following are the properties of the desired mapping:

1) If a node has the maximum fan-out (i.e., it has $2^k$ children), all the children of this node are mapped to contiguous table entries. The children are ordered according to the labels of the edges connecting them to their common parent. Thus the child with label "00...0" occupies the lowest address of the contiguous block and that with label "11...1" occupies the highest address.

2) If a node has less than $2^k$ children, the mapping of these children must preserve the same relative positions that would have been obtained if the node had maximum fan-out. For example, if k=3 and a node has three children whose edges have the labels "001", "011" and "110", then the three children should be mapped to entries r+1, r+3, and r+6, respectively, for some integer index r.

3) Only the mapping of nodes having a common parent need to obey the above rule. There is no restriction imposed on the relative locations of the two entries to which a node and its child are mapped. Also, there is no restriction on the position of the contiguous block (assigned to the children of some node) within the decode table.

4) To optimize the design (especially for VLSI and associative memory), the size M of the decode table must be minimized, i.e., M should be as close to N as possible.

The above special mapping of tree nodes into entries of the decode table will enable us to construct an efficient decoding operation with simple logic (as shall be explained shortly). First, we shall formulate the design of the k-bit decode table as a binary string mapping problem. The formulation is applicable to any tree-based codes (e.g., Shannon-Fano codes [SHAN49, FANO49], Fibonacci codes [LELE87], Huffman codes [HUFF52], etc.).

**Descendent Strings**

For each non-leaf node in the k-bit tree, we associate a binary string (called the descendent string) of length $2^k$. A bit in this string is set to 1 only if the index (position) of this bit is equivalent to the label of the edge leading to a child of this node. If the edge label for a child has less than k bits, extra zeros are appended to this label, at the least significant (rightmost) positions, to obtain a k-bit field that can be used as an index into the descendent string.

For example, if k=3 and a node has four children with edge labels "0" , "100", "101" and "11", the corresponding 8-bit descendent string is "10001110". Notice that the short labels "0" and"11" are first extended to become "000" and "110" then used to set the two bits at positions 0 and 6 of the string.

Remarks:

Descendent strings constructed as above satisfy the following:

1) Each child node corresponds to a unique 1 in the descendent string of its parent. Notice that appending zeros to short labels at the rightmost position (rather than the leftmost position) preserves this uniqueness.
2) The total number of 1's in all descendent strings is equal to N, the number of non-root nodes in the k-bit Huffman tree.

Since leaf nodes don't have children, they all have identical descendent strings of the form "00...0". It will be clear shortly that these descendent strings (all zeros) will not need to be considered in our search for the optimal mapping.

**Mapping of Descendent Strings**

Given the descendent strings of the non-leaf nodes of a k-bit Huffman tree, a binary string W is constructed such that

1) Each descendent string is mapped to $2^k$ consecutive bits in W.

2) Overlapping of descendent strings within W is permitted provided that no two bits having value 1 are mapped to the same position in W.
3) Each bit in W is covered by at least one descendent string, i.e., for each bit in W, there is at least one descendent-string bit that is mapped to it.
4) The value of each bit in W is obtained by performing the bitwise OR operation on the descendent-string bits that are mapped to it (notice that at most one of these descendent-string bits is allowed to have a value of 1).

If W is constructed as above, then the number of 1's in W is also equal to the number of non-root nodes N, i.e., each non-root node in the k-bit tree is associated with a unique 1 in W. Assume that after truncating any leading and trailing zeros from W, the resulting string, say S, is of size M bits. A decoding table of size M entries is then constructed. A non-root node in the k-bit tree is mapped to the entry of the decode table whose index is equal to the index of the unique 1 associated with this node in S. To optimize the design, the value of M should be minimum.

**The K-Bit Contiguous Binary Superstring (CBS) Problem**

We now summarize the mapping problem discussed above. The problem is a slightly different version of the one posed in [MUKH91a]. The formulation is applicable to any tree-based codes (e.g., Shannon-Fano codes [SHAN49, FANO49], Universal codes of Elias [ELIA75], Huffman codes [HUFF52], etc.). In this report, we shall concentrate on solving the problem for Huffman codes.

Instance: a collection of binary strings (descendent strings) of length $2^k$ bits each. Let N be the total number of 1-valued bits in all the descendent strings.

Problem: find a binary string $W=0^*S0^*$ where S is a minimum-length binary string, with no leading or trailing zeros, which has exactly N 1-valued bits such that every descendent string can be positioned (aligned) within W so that each 1-valued bit in any descendent string corresponds (is mapped) to a unique 1 in S.

In the above definition, the notation $0^*$ is used to denote an all-zero string of arbitrary length (possibly empty), and $0^*S$ is used to denote the concatenation of the two strings $0^*$ and S.

**Definition:**

The vacancy ratio of the solution for the CBS problem is defined as the ratio (M-N)/M and the expansion ratio is defined to be (M-N)/N, where M is the size of the resulting string S and N is the number of 1-valued bits in all the descendent strings.

**Example 2.1:**

For k=3, consider the three descendent strings

D1= "01010000" D2= "01001000" D3= "10000100"

The optimal solution in this case is S= "1111101" and W =0S00 = "0111110100". The starting positions (which we call the CBS indexes) of the strings D1, D2, and D3 within W are 2, 0, and 2, respectively. The 7-bit string S implies that we need to use a decoding table of M=7 entries. One entry in this table (the one before the last) is not used to store decoding information, but may be freely used to store any other data. The fraction of table entries that are not used is given by the vacancy ratio which, in this case, is equal to 1/7 . The expansion ratio of 1/6, on the other hand, gives the ratio between the extra (non-used) space to the original number of nodes N.

Notice that the descendent strings need not be distinct since several nodes (e.g., nodes 1 and 8 in Fig. 2.2) may have the same pattern of descendent edges. In general, solving the CBS problem seems to require exhaustive search, but sub-optimal solutions can be obtained using a variety of fast heuristic algorithms. The CBS problem has the flavor of some compute-bound string matching problems (e.g., the superstring problem [TARH88]) but is quite distinct from them. We conjecture that the general k-bit CBS problem is NP-hard. Proving this conjecture is posed as an open problem. Fortunately, the special case of Huffman's decoding offers some useful properties that help tackle the CBS problem. For example, descendent strings in 2-bit Huffman trees have only 4 valid patterns (out of 16 distinct ones) and those in 3-bit Huffman trees have only 25 valid patterns (out of 256 distinct ones). In addition, we are particularly interested in the case of k=2 since it represents the most suitable and viable value for associative memory and hardware implementations. We

shall therefore concentrate on solving the CBS problem for 2-bit Huffman trees.

**Lemma 1:**
For 2-bit Huffman trees, the descendent strings have only four 4-bit patterns: "1111", "1110", "1011", and "1010".

Proof of this lemma is based on the simple observation that every non-leaf node in a 1-bit Huffman tree must have two children. Consequently, every non-leaf node in a 2-bit Huffman tree must have either four children (pattern "1111"), three children (patterns "1110" and "1011") or two children (pattern "1010"). Other patterns (e.g., "1000", "0101") are not possible because of the above property of Huffman tree and the method used to append zeros to short labels during the construction of descendent strings. Notice that Lemma 1 implies that there will be no leading zeros in the string W used in the definition of the CBS problem.

Based on the above lemma, we can now construct an optimal algorithm for solving the 2-bit CBS problem for Huffman trees. The idea is to cluster the descendent strings into four groups. Group G1 contains all strings of value "1111". These strings are simply placed consecutively, one after the other, on 4-bit contiguous fields. The second group, G2, contains strings of the form "1110". Again, we map these strings onto 4-bit fields such that the first bit of a field overlaps with the last bit of the previous field. The third group G3 and the fourth group G4 contain strings of the form "1011" and "1010", respectively. First, we repeatedly try to pair one string from G4 with one string from G3 (shifted one bit to the right) and map them onto a 5-bit field . Next, we repeatedly try to pair two G4 strings onto the 4-bit field "1111" (this is done by shifting one string one bit to the right and then discarding its trailing zero). Finally, any remaining strings are mapped separately onto 4-bit consecutive fields. A high-level description of the algorithm is given below.

**Algorithm CBS_2H;**

Input: a collection of 4-bit descendent strings (not necessarily distinct).

Output: binary string S and the CBS index (starting position within S) to which each descendent string is mapped.

Method:

```
Cluster the input strings by pattern into 4 groups
S = Null;  i = 0   /* initialization */
For each string D in G1 do          /* D = "1111" */
        map D to i
        append "1111" to S
        i = i + 4; endfor;
For each string D in G2 do          /* D = "1110" */
        map D to i
        append "111" to S
        i = i + 3; endfor;
While (both G3 and G4 are not empty) do
        remove a string D from G4     /* D ="1010" */
        map D to i
        remove a string D' from G3     /* D' ="1011" */
        map D' to i+1
        append "11111" to S
        i = i + 5; endwhile;
/* now at least one of G3 and G4 is empty */
While (G4 has at least two members) do
        remove two strings D and D' from G4    /*  D=D'="1010" */
        map D to i and D' to i+1
        append "1111" to S
        i = i + 4; endwhile;
If (G4 is not empty) do
        remove last string D from G4      /* D = "1010" */
        map D to i
        append "101" to S
        i = i + 3; endif
For each remaining string D in G3 do;  /* D = "1011" */
        map D to i
        append "1011" to S
        i = i + 4; endfor
end CBS_2H;
M = i   /* M is the size of string S */
```

**Example 2.2:**

For the 2-bit Huffman tree of Fig. 2.2, the number of non-root nodes is N=17 and the descendent strings of the six non-leaf nodes are as follows:

Node 0: $D_0$ = "1111"

Node 1: $D_1$ = "1010"

Node 2: $D_2$ = "1011"

Node 4: $D_4$ = "1110"

Node 8: $D_8$ = "1010"

Node 11: $D_{11}$ = "1011"

In this case, algorithm CBS_2H produces a string S of 17 consecutive 1's and generates six CBS indexes that map $D_0$ to 0, $D_4$ to 4, $D_1$ to 7, $D_2$ to 8, $D_8$ to 12, and $D_{11}$ 13. The vacancy ratio of this mapping is zero.

**Lemma 2:**

For 2-bit Huffman trees, the linear-time algorithm CBS_2H is optimal, i.e., it produces a string S of minimum length.

Proof of the above lemma can be established by considering the four patterns of valid descendent strings in 2-bit Huffman trees. Notice that the algorithm produces compact mapping (without any expansion) for patterns "1111" and "1110" as well as for "1011"/"1010" and "1010"/"1010" string pairs. The only expansion introduced by the algorithm is due to either

i) a single (left-over) string of value "1010", or
ii) Strings of value "1011" which are in excess of their "1010" counterparts.

Notice that at most one type of expansion (i or ii above) can occur for any given 2-bit Huffman tree. It is easy to see that such expansion (if it occurs) is necessary. In other words, any valid mapping will produce a string S with a number of 0's equal to or greater than the number of left-over strings causing the expansion.

**Lemma 3:**

The worst case vacancy ratio for algorithm CBS_2H is 0.25 (corresponding to a worst case expansion ratio of 1/3).

It is easy to see that this worst case ratio can only be obtained if all descendent strings have value "1011". Notice that in this case, there is only one valid mapping that can be used to solve the CBS problem. This worst case is, however, highly unlikely. Typical values of the vacancy ratio for the optimal mapping are usually much smaller (and are often zero) due to the pairing of strings in G3 and G4.

**The 2-Bit Decoding Process**

We now turn back to the problem of 2-bit Huffman's decoding. For the tree of Fig. 2.2, a decode table of 17 entries (numbered 0 through 16) is used to store the non-root nodes of the tree. The root is stored in a separate block (core-resident global variable) to allow faster access to it. The mapping of a tree node into an entry in the decode table is obtained by adding the following two components: i) the CBS index of the descendent string of the parent of this node and ii) the label of the edge connecting this node to its parent. As explained before, labels of length one bit are extended to 2 bits by appending a rightmost zero. For example, node 6 (symbol B) in Fig. 2.2 is mapped to entry # 9 in the decode table; this is obtained by adding the CBS index of string $D_1$ obtained in Example 2.2 (i.e., decimal value 7) to the extended label "10" (decimal value 2).

A field in the decode table, called "base", is used to store the CBS index for each non-leaf node. Recall that for the tree of Fig. 2.2, these CBS indexes are as follows

| node # | 0 | 1 | 2 | 4 | 8 | 11 |
|---|---|---|---|---|---|---|
| CBS index | 0 | 7 | 8 | 4 | 12 | 13 |

In the case of leaf nodes, the "base" field is used to store the output code of the corresponding symbol. We assume that the value of "base", or a flag bit in it, can be used to determine whether the corresponding node is a leaf or not (alternatively, a separate Boolean flag can be used). The basic loop of the decoding process proceeds as follows:

a) read two bits from the compressed file,

b) add these two bits, treated as a 2-bit integer, to the value of the "base" field of the current node

c) the result gives the index of the node to be visited next.

There is now one last issue that needs to be solved, namely, handling short labels. The last edge leading to a leaf node may have a label of length one bit (rather than 2). In this case, we should only use the first input bit (appended with 0) to complete the current decoding process. The other (non-used) input bit should be attached to a new bit from the compressed file and the resulting two bits are then used to start a new decoding operation from the root of the 2-bit tree. To accomplish this, two Boolean flags $f_0$ and $f_1$ in the decode table are used to indicate short labels as follows: if the next input bit has a value j and flag $f_j$ has a value of 1 (True), then an edge with a short label is encountered. For example, the value of the two flags $(f_0,f_1)$ for nodes 2, 4, and 8 of Fig. 2.2 are (1,0), (0,1), and (1,1), respectively. The two flags are not needed for leaf nodes; their values in this case are immaterial. Table 2.1 shows the decode table for the tree of Fig. 2.2 based on the optimal mapping obtained in Example 2.2. The decode table has 17 entries (numbered 0 through 16); each entry contains the three fields: base, $f_0$, and $f_1$. For clarity, Table 2.1 also gives the sequential index of each table entry as well as the index (node #) of the tree node assigned to that entry. These latter two fields are included for the purpose of clarification; they are not actually stored in the decode table. The root node is stored in a separate global entry with values 0 for base, 0 for $f_0$, and 0 for $f_1$.

Algorithm Decode_2H gives a high-level description of the 2-bit Huffman's decoding algorithm. The algorithm uses a decode table denoted DT and a separate root entry as explained before. The auxiliary variables Base, F[0], and F[1] are used to store the base, $f_0$, and $f_1$ fields, respectively, of the current node while the variables v[0] and v[1] are used to hold the two input bits currently being processed.

Table 2.1. Decode table for the tree of Fig. 2.2.

| node # (see Fig. 2.2) | entry index | base | $f_0$ | $f_1$ |
|---|---|---|---|---|
| 1 | 0 | 7 | 1 | 1 |
| 2 | 1 | 8 | 1 | 0 |
| 3 | 2 | G | | |
| 4 | 3 | 4 | 0 | 1 |
| 10 | 4 | H | | |
| 11 | 5 | 13 | 1 | 0 |
| 12 | 6 | L | | |
| 5 | 7 | A | | |
| 7 | 8 | C | | |
| 6 | 9 | B | | |
| 8 | 10 | 12 | 1 | 1 |
| 9 | 11 | F | | |
| 13 | 12 | D | | |
| 15 | 13 | I | | |
| 14 | 14 | E | | |
| 16 | 15 | J | | |
| 17 | 16 | K | | |

```
Algorithm  Decode_2H;
/* 2-bit Huffman's decoding */
while (not end of file) do
      initialize Base, F[0], and F[1] from root entry
      Repeat
            read enough input data and
            store one input bit into v[1]
            if ( F[v[1]] = 1 )
            then v[0] := 0   /* short label */
            else store another input bit into v[0] endif;
            offset := integer {v[1]v[0] } /* form a 2-bit integer */
            Next := Base + Offset
            Base := DT[Next].base
            if (Base is not a symbol)
            then {F[j] := DT[Next].f_j   for j=0,1}
            else {Output the symbol stored in Base} endif;
      Until (Base is a symbol)
endwhile;
```

## Implementation

A prototype multibit Huffman's encoder/decoder chip has been fabricated for k=2. The various simulation experiments that we have conducted as well as the time analyses and evaluation tests of this chip indicate that the 2-bit hardware

approximately doubles the throughput of the decoder (compared to the original Huffman's hardware [MUKH91b]). The prototype chip, called MARVLE, uses 2-micron CMOS technology, has a 512x12 static RAM with an access time of 4 nanoseconds and consists of 49,695 transistors. The VLSI hardware is very suitable for real-time applications and can also be used to implement the JPEG baseline compression scheme.

## Multigroup Compression

The multigroup scheme is tailored to take advantage of the property of symbol reference locality. By modifying the Huffman's algorithm to take advantage of this property, both the compression ratio and the decoding time can be significantly improved (at the expense of some additional encoding overhead). In this report, we present a multigroup decoding algorithm that works for one-level and two-level hierarchies having arbitrary number of groups. We shall first illustrate the basic idea of the multigroup approach by an example, then proceed to discuss other relevant aspects and variations.

**Example 2.3:**

Assume that the set of input symbols consists of twelve members as shown in Fig. 2.1 and consider a relational scheme with three attributes whose values are obtained from three different types of fixed-length domains. The first domain, DOM1, is of length 13 and is restricted to the five symbols A, B, C, D, and E. The second domain, DOM2, is of length 12 and is restricted to the three symbols F, G, and H. The four remaining symbols I, J, K, and L are used in DOM3 which has a length of 7. Furthermore, assume that the relative counts of the twelve symbols are 4, 3, 4, 1, 1, 2, 8, 2, 1, 1, 1, 4 (same counts used to construct the tree of Fig. 2.1). Thus the string

$$S_1 = A^4B^3C^4DEF^2G^8H^2IJKL^4$$

is a valid tuple (record) that satisfies both the above relative counts and the locality of symbol occurrences within the three attributes (the notation $A^j$ is used to indicate that symbol A is repeated j times). Using the Huffman tree of Fig. 2.1, the string $S_1$ can be encoded using 104 bits. However, better results can be obtained if we split the tree of Fig. 2.1 into three groups and provide a mechanism to switch among these groups during the encoding and decoding

processes. Fig. 2.3 gives a multigroup design corresponding to the Huffman tree of Fig. 2.1. The scheme uses a two-level hierarchy of Huffman trees. The first level contains three group trees corresponding to the symbols of the three domains (labels inside leaf nodes in the group trees represent the frequency counts of these nodes). In each group tree, we introduce an extra symbol, denoted by "@", which we call the switch indicator. The code of this symbol is used (by the encoder) to inform the decoder that the next symbol belongs to a different group tree. The encoder then indicates the identity of the new group by emitting the appropriate code from the switch tree at the second level of the hierarchy.

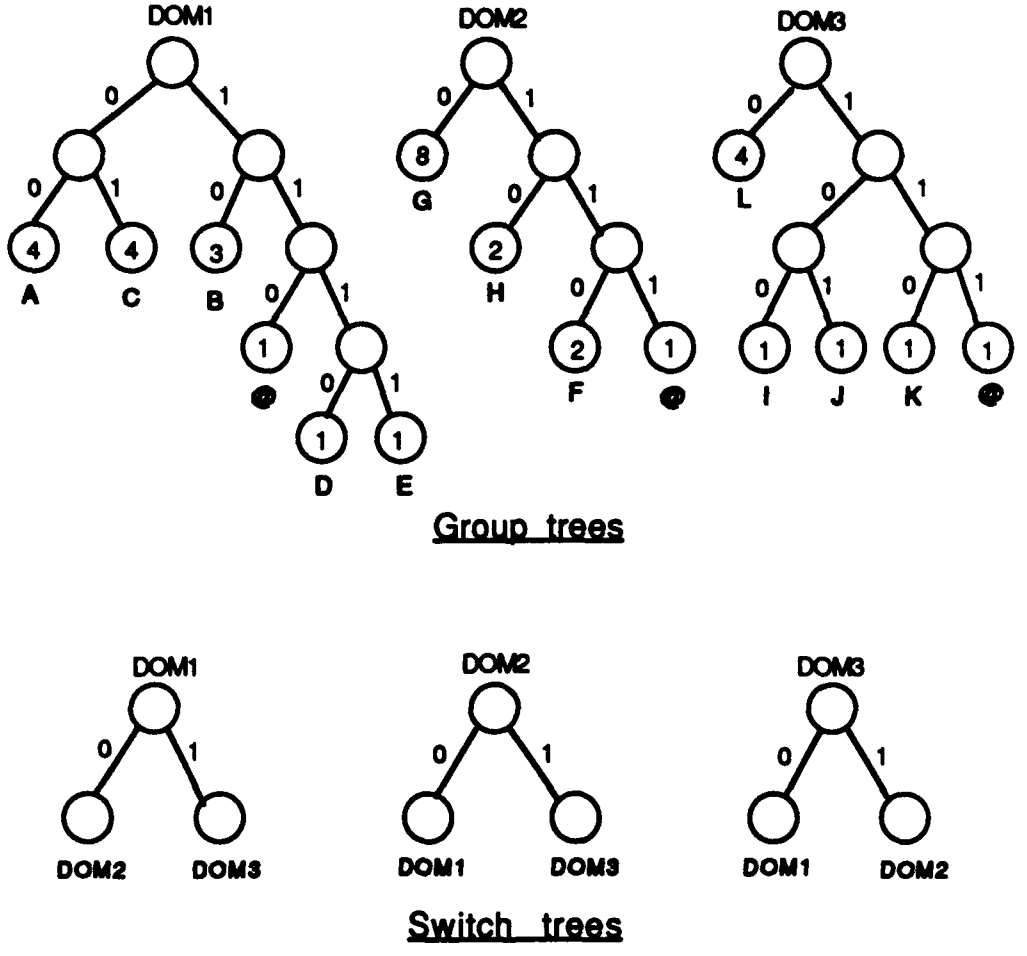



Fig. 2.3. Multigroup scheme for the set of Fig. 2.1

For example, when the string "EFG" is encoded, the following sequence of bits is produced

```
1111        /* 4-bit code of E */
110         /* 3-bit code of @ in the DOM1 group tree */
0           /* code of DOM2 in the DOM1 switch tree */
110         /* 3-bit code of F */
0           /* code of G */
```

Using the multigroup scheme of Fig. 2.3 and assuming the encoder and decoder are initialized to use DOM1 as the starting group, the string $S_1$ is now encoded in 69 bits. This represents an improvement equal to (104 - 69)/104 or 33% over the Huffman's scheme of Fig. 2.1. If DOM2 (or DOM3) is used as the starting group, string $S_1$ is encoded in 73 bits (30% improvement).

Remarks:

1) If the multigroup scheme uses m groups, each switch tree in the second level will have m-1 leaf nodes. Statistics about the transition from one group to the other should be collected to establish the correct weights needed to build these trees as proper Huffman trees. For the case of m=3, the switch trees have the fixed two-node topology shown in Fig. 2.3 (regardless of the values of the transition frequencies). In the special case of m=2, the switch trees are eliminated; the symbol @ in each group tree simply indicates that the next symbol belongs to the other group tree. The original Huffman scheme (Fig. 2.1) can be viewed as the special case of m=1.

2) The relative wight assigned to the symbol @ in each group tree should be based on the frequency of switching from that group to others, i.e., should be based on the average number of consecutive symbols (from that group) appearing in the input before a switch to another group occurs. Notice that there is no ambiguity introduced when the symbol @ is assigned different codes in the different group trees.

3) The set of input symbols does not have to be partitioned into disjoint subsets. Rather, some symbol(s) may be included in two or more group trees. As in the case of the switch indicator @, such symbols may have different codes in the different group trees without causing any ambiguity. In practice, a

useful application of this strategy is to include the blank character in both the digit and the alphabet group trees.

4) Although the encoding phase of the multigroup scheme is more complex than its Huffman counterpart, the decoding logic is essentially the same, namely, the input bits are used one at a time to traverse a tree structure. If the multigroup scheme is appropriately applied to files having the property of symbol reference locality, the improved compression ratio means that the bit-serial decoder will need to operate on less number of bits and can therefore be significantly faster. For example, a compression improvement of 33% (as in the above example) would typically translate to an improvement in the decoding speed of about 20% compared to the original Huffman scheme.

**The Decoding Algorithm:**

Algorithm MG_DECODE, given below in pseudo-code, is a high-level description of the multigroup decoding operation. The algorithm works for any number of groups m $\geq$ 1 and handles both group and switch trees using the same loop statement.

```
MG_DECODE,
/* Initialize pointers */
Current_root := root of first group tree;
Ptr := Current_root
while (not end of file) do;
      If (Ptr points to a  non-leaf node)
      then  {
            case
            :input bit = 0:       Ptr := left_child[Ptr]
            :input bit = 1:       Ptr := right_child[Ptr]
            endcase;
            }
      else  {
            If (Contents[Ptr] is a symbol)
            then {Output this symbol; Ptr := Current_root}
            else  Ptr := Current_root := Contents[Ptr]
            endif
            }
      endif;
endwhile;
```

Notice that a leaf node may be used to store either the code of an output symbol or an address to another node (i.e., the address of the root of a new tree). Specifically, a leaf node in a switch tree should be used to store the address of the root node of a group tree. Similarly, the leaf node corresponding to the symbol @ in a group tree should store the address of the root of the corresponding switch tree. Other leaf nodes in the group trees are used to store the code of output symbols.

As mentioned earlier, the second-level (switch) trees are eliminated in the special case of m=2. This concept, however, can be extended to other higher values of m by introducing additional switch-indicator symbols in each group tree. We call the resulting scheme the one-level multigroup scheme. For example, if the leaf node of the switch indicator @ in the first group tree (i.e., group DOM1) of Fig. 2.3 is replaced by the corresponding switch tree, we obtain the modified group tree shown in Fig. 2.4. The switch indicator @ is thus replaced by two nodes (called the direct switch nodes) which store the addresses of the roots for the second and third group trees. These latter trees are modified in the same fashion to obtain a single level of group trees.

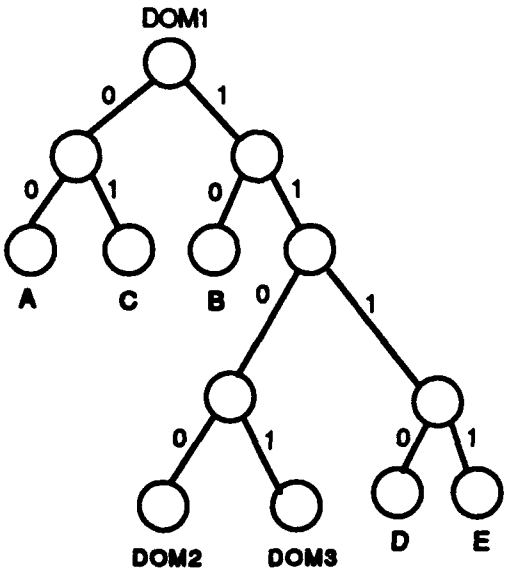



Fig. 2.4. An example one-level group tree

It is obvious that the particular modification shown in Fig. 2.4 is exactly equivalent to the two-level scheme of Fig. 2.3 in the sense that they both produce the same compressed output for any string. The single-level scheme, however, has the flexibility of adjusting the position of the direct switch nodes (based on the transition statistics) in order to further improve the compression ratio. The following example illustrates this point.

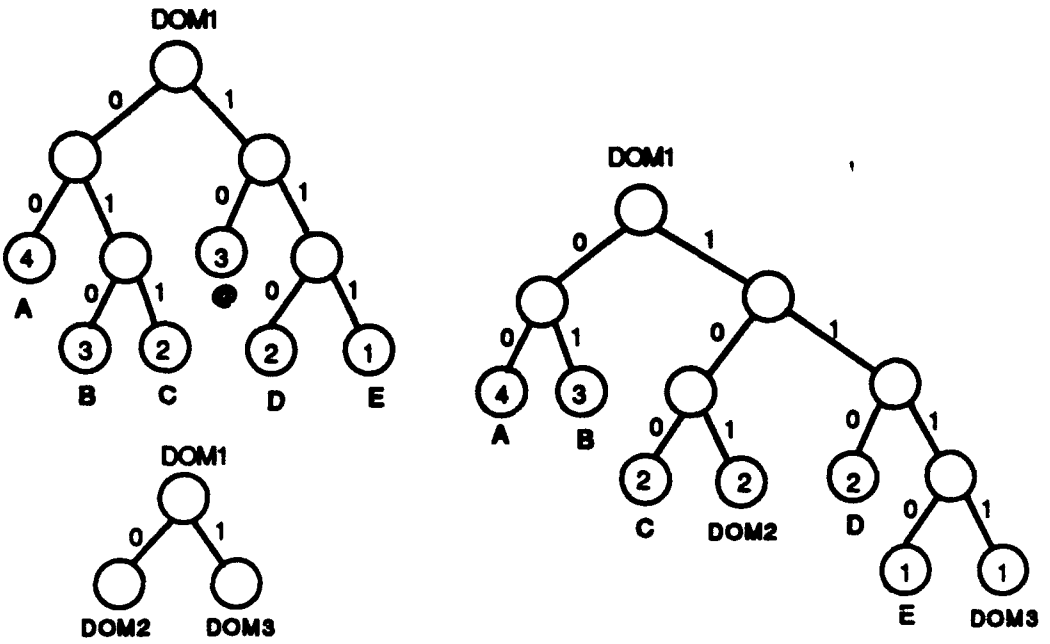
**Example 2. 4:**

Consider again the first group, DOM1, of the multigroup scheme of Fig. 2.3 but with the following modified statistics:

a) the relative counts (weights) of the symbols A, B, C, D, and E are now 4, 3, 2, 2, and 1 respectively.
b) the average number of consecutive symbols from group DOM1 before a switch to another group occurs is four.
c) the frequency of transitions from DOM1 to DOM2 is double that from DOM1 to DOM3.

Based on the above statistics, the switch indicator @ in the two-level multigroup scheme will have a relative count of 3 as shown in Fig. 2.5-a. The single-level scheme shown in Fig. 2.5-b, however, uses two direct switch nodes: DOM2 and DOM3 with relative weights 2 and 1, respectively.

Assuming the modified statistics hold, the one-level scheme of Fig. 2.5-b gives a 2.5% average improvement in the compression of symbols from the first group than its two-level counterpart. The Huffman trees of Fig. 2.5 are obviously not unique, but all valid Huffman trees constructed for the same weights would surely give the same compression ratio.




a) two-level scheme for DOM1 b) one-level scheme for DOM1

Fig. 2.5. Multigroup schemes for modified statistics

**Lemma 4:**

For the same input weights, the one-level multigroup scheme gives equal or better compression ratio than that obtained by the two-level scheme.

The single-level scheme, however, requires the availability of accurate statistics about the transition frequency from a local group to each other local group. These statistics are needed in order to assign proper weights to the direct switch nodes (relative to the original symbols). In contrast, group trees in the two-level scheme only require aggregate information about switching from a group. Specifically, only the average number of consecutive input symbols from a group is needed to determine the relative weight of the switch indicator @ in that group tree. Information about individual transitions between pairs of groups, however, is used in the switching (second level) trees; but high accuracy about these transition frequencies may not be at all needed. For example, the case m=3 gives a unique topology of switching trees and no information about individual transition frequencies is needed. Similarly, the case m=4 would only require information about the ordering of

the individual transition frequencies and not their relative values (i.e., the topology of each Huffman switch tree in this case is completely defined if we know which one of its three leaf nodes has the highest weight).

It is interesting to mention that both the single-level and two-level multigroup options can be used simultaneously within the same scheme. In other words, individual groups can be made single-level or two-level based on the availability of transition frequencies. Basically, the resulting hybrid scheme does not change the logic or increase the complexity of the multigroup method. We conclude this section by the following observations.

Observation 1:

Algorithm MG_DECODE handles any number of groups m $\geq$ 1 and operates correctly for the two-level scheme, the one-level scheme, or any combination of these two schemes. The algorithm is a generalization of Huffman decoding; running the algorithm with m=1 would reduce to the original Huffman scheme (with the same compression result and almost the same time overhead).

Observation 2:

The multibit scheme can be incorporated into the multigroup technique to further enhance the speed of the decoding process. For example, the group trees of Fig. 2.3 can be changed into corresponding 2-bit decode trees. The resulting multibit multigroup scheme would improve the compression ratio and significantly improve the decoding speed over the original Huffman scheme.

## Application to Arithmetic Coding

Our discussion so far has concentrated on tree-based codes (primarily Huffman codes). Attempting to extend the two techniques discussed in this report to the case of arithmetic coding [WITT87] would reveal the following:

a) The multigroup technique is straightforwardly applicable to arithmetic coding. When used with arithmetic coding, the multigroup scheme roughly gives the same compression benefit as that obtained for the Huffman's

scheme. Unlike the bit-serial Huffman decoding, however, the speed of arithmetic decoding is not significantly improved by the application of the multigroup scheme.

b) Arithmetic coding does not emit a separate code for each input symbol; the decoding process is not bit-serial. The multibit decoding scheme is therefore not applicable to arithmetic coding.

In what follows, we briefly discuss the application of the multigroup scheme to arithmetic coding, using the case of m=2 as an example.

The idea of arithmetic coding is to map a message into an interval of real numbers between 0 and 1. Consider the set of symbols $\Gamma = \{A_1, A_2, \ldots, A_V\}$ where the probability of occurrence of symbol $A_k$ is given by $p_k$. In arithmetic coding, each symbol is assigned an interval proportional to its probability of occurrence. The interval for symbol $A_k$ is denoted by $[a_k, b_k)$ and is computed as follows

$$a_1 = 0$$

$$a_k = \sum_{j=1}^{k-1} p_j \qquad 2 \le k \le V$$

$$b_k = a_k + p_k \qquad 1 \le k \le V$$

Thus symbols are assigned nonoverlapping intervals whose union is the interval [0.0,1.0). The idea of arithmetic coding is to start with the initial interval [0.0,1.0) and then narrow it repeatedly (as symbols are processed) such that each interval is totally contained in the preceding one. In general, if $I_n = [s_n, f_n)$ is the current interval, and the next symbol is $A_j$ with range $[a_j, b_j)$, then the next interval $I_{n+1} = [s_{n+1}, f_{n+1})$ is computed as follows.

$$s_{n+1} = s_n + a_j * (f_n - s_n)$$

$$f_{n+1} = s_n + b_j * (f_n - s_n)$$

This assures that $I_{n+1}$ is a subinterval of (i.e., totally contained in) the interval $I_n$. Furthermore, the following relationship holds true.

$$\text{length}(I_{n+1}) = (b_j - a_j) * \text{length}(I_n)$$

where length $(I_n) = f_n - s_n$. It is easy to see that symbols with higher probabilities' of occurrence (and hence with larger intervals) have slower effect on narrowing the interval than symbols with smaller probabilities [WITT87]. Assigning larger intervals to the most frequent symbols would therefore increase the compression efficiency since it enables encoding more symbols (on the average) in the same fixed-length field. The multigroup approach can be applied to arithmetic coding using the same principles used in Huffman's encoding. We shall briefly cover the case of m=2 as an illustration. Without loss of generality, assume that the set of symbols $\Gamma$ is partitioned into the following two sets:

$$\Gamma_1 = \{A_1, A_2, \ldots, A_\mu, @\}$$
$$\Gamma_2 = \{A_{\mu+1}, A_{\mu+2}, \ldots, A_\nu, @\}$$

where @ is the switch indicator as explained before. Assuming that symbols $A_1$ through $A_\mu$ tend to occur consecutively in groups of expected length of $L_1$, the new (adjusted) probabilities of occurrence for symbols in the set $\Gamma_1$ are computed as follows:

$$q_k = \frac{p_k}{\sum_{j=1}^{\mu} p_j} * \frac{L_1}{L_1 + 1} \qquad 1 \le k \le \mu$$

The probability of the switch indicator @ in $\Gamma_1$ is given by

$$q_@ = \frac{1}{L_1 + 1}$$

The modified probability of a symbol in the set $\Gamma_1$ is larger than its original value (i.e., $q_k > p_k$) if the following condition is satisfied.

$$\left( \sum_{j=1}^{\mu} p_j \right) * \left(1 + \frac{1}{L_1}\right) < 1$$

In that case, the length of the interval of symbol $A_k$ in the set $\Gamma_1$ is larger than that of its counterpart in the original set $\Gamma$. This means that symbol $A_k$ in the new scheme will have a slower narrowing effect than in the original arithmetic coding scheme. If the value of $L_1$ is not very small, the extra narrowing effect produced by the symbol @ (at each locality switch from the

set $\Gamma_1$ to the set $\Gamma_2$) is more than offset by the increase in the ranges of the individual original symbols in $\Gamma_1$. The modified probabilities for symbols and the switch indicator in the set $\Gamma_2$ are given by

$$q_k = \frac{p_k}{\sum_{j=\mu+1}^{\nu} p_j} * \frac{L_2}{L_2 + 1} \qquad \mu+1 \leq k \leq \nu$$

$$q_{@} = \frac{1}{L_2 + 1}$$

where $L_2$ is the expected length of sequences of symbols $A_{\mu+1}$ through $A_\nu$ appearing consecutively. Similar remarks apply to the symbols in this group as those discussed for the set $\Gamma_1$.

In summary, we have discussed two schemes for enhancing the Huffman's decoding. The multibit scheme reduces the time overhead of the bit-serial decoding operation. The case of 2-bit decoding is quite attractive for practical implementation; the report presented an optimal solution for the 2-bit CBS problem. Further research is needed to investigate the practicality of other k-bit decoders and to determine the value of k for which a k-bit decoder represents the best tradeoff between the speed of decoding and logic (or hardware) complexity. The multigroup scheme is suitable for files that exhibit the property of locality of symbol references. The scheme improves the Huffman's compression efficiency as well as the time overhead of the decoding process. The report presented a multigroup decoding algorithm that works for one-level and two-level hierarchies with arbitrary number of groups. The multibit scheme can be incorporated into the multigroup technique to further enhance the speed of the decoding process.

## References


[BASS91] Bassiouni, M. and Tzannes, N. "Integrated lossless/lossy schemes for image compression" Proc. of IEEE International Conf. on Acoustics, Speech, and Signal Processing, May 1991.

[BASS85] Bassiouni, M. "Data compression in scientific and statistical databases" IEEE Transactions on Software Engineering, Vol. SE-11, No. 10, October 1985, pp. 1047-1058.

[ELIA75] Elias, P. "Universal codeword sets and representations of the integers" IEEE Trans. on Information Theory, Vol. 21, No. 2, 1975, pp. 194-203.

[FANO49] Fano, R. *Transmission of information*. MIT Press, Cambridge, MA, 1949.

[HUFF52] Huffman, D. "A method for the construction of minimum redundancy codes" Proc. IRE, Vol. 40, 1952, pp. 1098-1101. pp. 435-447.

[LELE87] Lelewer, D. and Hirschberg, D. "Data compression" ACM Computing Surveys, Vol. 19, No. 3, 1987.

[MUKH91a] Mukherjee,A; Bheda, H; Bassiouni, M. and Acharya, T. "Multibit decoding/encoding of binary codes using memory based architectures" Proc. of IEEE Data Compression Conference, April 1991, pp. 352-361.

[MUKH91b] Mukherjee, A.; Ranganathan, R. and Bassiouni, M. "Efficient VLSI designs for data transformation of tree-based codes" IEEE Transactions on Circuits and Systems" Vol. 38, No. 3, March 1991, pp. 306-314.

[SHAN49] Shannon, C. and Weaver, W. *The Mathematical Theory of Communication*. University of Illinois Press, 1949.

[TARH88] Tarhio, C. and Ukkonen, E. "A greedy approximation algorithm for constructing shortest common superstrings" Theoretical Computer Science, Vol. 53, 1988.

[WELC84] Welch, T. "A technique for high-performance data compression" Computer, Vol. 17, No. 6, 1984, pp. 8-19.

[WITT87] Witten, I.; Neal, R. and Cleary, J. "Arithmetic coding for data compression" Communications of ACM, Vol. 30, No. 6, June 1987, pp. 520-540.

# Appendix I

Copy of paper entitled

## Memory Based Multibit CODEC Architecture

*(submitted for publication)*

# Memory Based Multibit CODEC Architecture


## Amar Mukherjee
## Tinku Acharya
## M. A. Bassiouni

**Department of Computer Science**

**University of Central Florida**

**Orlando, Florida 32816.**

**U.S.A.**

**(407)-823-2763**

*amar@eola.cs.ucf.edu*

*acharya@eola.cs.ucf.edu*



**Abstract**

We present a new memory based CODEC architecture to design a special purpose hardware for real-time multibit compression/decompression of binary data. The proposed architecture is based on a novel idea of mapping the decoding/encoding tree of any variable length binary code on to a memory device that corresponds to simultaneous decoding/encoding of multiple bits. The hardware is programmable, easily adaptable and yields a high compression rate. A prototype 2-micron VLSI chip based on this architectural idea has been designed. This chip occupies a silicon area of 6.9 x 6.8 square millimeters and it contains 49,695 transistors with estimated compression rate of 88 Mbits/sec and a decompression rate of 53 Mbits/sec with a clock rate of 50 MHz. The algorithms have been tested with different types of variable-length binary codes including the JPEG baseline compression scheme.

Associated with the memory map, a new binary string alignment problem, called the **Contiguous Binary SuperString** (CBS) problem is formulated and heuristic algorithm is developed to solve it. An efficient algorithm for this problem is posed as an open question.



**Keywords:** CODEC, compression, decompression, JPEG, Multibit Data Compression/Decompression, tree based code, reverse code, reverse binary tree, memory map, perfect map, Contiguous Binary Superstring, CBS.


## 1 Introduction

The primary objective of data compression algorithms is to reduce the redundancy in data representation in order to decrease data storage requirement. Reducing the storage require-

ment is equivalent to increasing the capacity of storage medium. In systems with levels of storage hierarchy, it may then be possible to store data at a higher (and faster) level thereby reducing the load on the I/O channels. Data compression offers an attractive approach to reduce the communication cost in transmitting exceptionally high volumes of data over long-haul links via higher effective utilization of the available bandwidth in the data links. The number of applications that require storage and transmission of large volumes of data is steadily increasing. Communication and display technologies allow the use of pictorial information and photographic images in various scientific, industrial, medical and consumer applications. Because of the large amount of data required to represent an image[1], compression techniques that expoloit redundancy in data are required for efficient transmission and storage. With respect to transmission of data, the NREN (Nastional Research and Educational Network) has characterized several networking applications (video teleconferencing , interactive visualization, composite imaging, etc.) to require peak bandwidth rate of 1 Mbits/Sec to 1000 Mbits/sec [BROM91]. Even with the advent of gigabit per second networks (to be developed by **CNRI, Corporation of National Research Initiatives** jointly with **NSF** and **DARPA** support), the development of efficient compression techniques in order to achieve high utilization and bandwidth will continue to be a design challenge for future communication systems. As an example, by 1995 NASA expects to acquire space and earth science data from spaceborne sensors which will amount to 28,000 gigabytes of data in its archive [GREE88]. Achieving real-time linkage among geographically-distant LAN (local area network) sites is one of the major technical challenges facing the implementation of long-haul data communication networks. In order to handle such staggering amounts of data, application-specific hardware algorithms and custom **VLSI** chips for data compression have to be developed as standard components for communication and storage.

A vast amount of literature is available on data compression techniques [LELE87][2]. Data compression techniques could be lossless or lossy. The lossless methods can recover an exact copy of the original data from the compressed data whereas the lossy techniques allow the decompressed data to be an approximation of the original data. The lossy techniques are usually applicable to image data where transform and other techniques [WALL90, NETR88, ARPS88, CLAR85] have been used to produce compression ratios of about

[1]Still pictures: ISO JPEG standards, 4.97 Mbits per picture frame; Motion pictures; CCIR 601 (4:3:2, NTSC) standard, 169.92 Mbits/sec with 30 frames/sec; Visual telephony, CCITT px64K standard, 12.165 Mbits/sec with 10 frames per second [NETR88, HANG90].

[2]There are also a number of excellent books [STOR88,NETR88,GW87] that treat compression techniques and image processing technology.

100:1. With the advent of VLSI technology, hardware support for lossy methods and special purpose VLSI chips are fast becoming standard components for image processing systems [CCUB90, VENB91, VETT86, SUN89][3]. The lossless methods have been traditionally used for large scientific and text databases and usually yield a compression ratio between 2:1 to 3:1. The classical lossless encoding methods have used tree-based codes which represent a large class of variable-length encoding schemes such as Huffman codes [HUFF52], Shannon-Fano Codes [FANO49, SHAN49], Universal codes of Elias [ELIA75], the Fibonacci codes [FRAE85], etc. The code set is represented by a tree in which the leaf nodes represent the symbols to be coded. The sequence of 1's and 0's in the unique path from the root of the tree to each leaf node represents the unique code for the corresponding symbol. The arithmetic codes [ABRA63, WITT87], the LZ codes [ZIVL78] and its several variants and the run-length code [GREE88, BASS85] are not tree-based codes and provide good compression ratios in many applications. In the absence of a suitable model of data to be compressed, the arithmetic and LZ methods provide better adaptive codes. The lossless methods in combination with lossy methods have been used in some image applications. For example, the baseline system proposed by ISO-JPEG [WALL90], an international still image standard, recommends the use of Huffman coding or arithmetic coding to encode the compressed image (obtained after transform and quantization steps) to further exploit its redundancy. Lossless methods are also used in specialized applications such as medical imaging (for diagnosis of disease) or satellite photography (such as level 0 or level 1A space image data of NASA [MILL88]) where reliability of reproduction of images is a critical factor.

In recent years, several special-purpose VLSI chips and architectures have been proposed to implement lossless compression algorithms. A class of parallel algorithms for compression by textual substitution is proposed in [STOR82, GONZ85, STOR88] and a hardware system consisting of several VLSI chips implementing their algorithm has been built [STOR90]. A hardware scheme implementing a variation of the LZ agorithm called the LZW algorithm was described in [WELC84]. Another realization of Ziv and Lempel's LZ2-type compression in hardware is described in [BUNT90].

Zito-Wolf has proposed VLSI architectures for the LZ1-type scheme [WOLF90a] using a binary tree and a linear systolic array that maintains the dictionary. The Hewlett-Packard's HP7980XC tape drive uses real-time data compression scheme to provide an extended per-

[3]These are typical references. There are a large number of other important references not cited here in order to conserve the size of this paper.

formance to the 6250 GCR format. A HCMOS VLSI chip for data compression based on a general-purpose adaptive binary arithmetic coding architecture was implemented by Arps et. al [ARPS88]. A set of VLSI chips have been built implementing the Rice algorithm [RICE90] at the NASA Space Engineering Research Centre for compressing satellite images and the hardware implementation is discussed in [VENB91]. The Rice algorithm is a lossless compression method which handles different entropy conditions by utilizing multiple coders, each of which is tuned to compress data a particular entropy range and selects the output from the coder that gives the best compression efficiency.

In this paper, we present a new memory based architecture for the design of special-purpose hardware for real-time compression and decompression of data. The architecture is suitable for any tree based codes and uses memory as its major component which can be very easily implemented in VLSI. The hardware algorithm is designed for parallel decoding and encoding of k bits of the code in one memory cycle. The details for the design for k=2 are presented. Compared to the case of k=1 (single bit decoding/encoding), increasing the value of k increases the average throughput by a factor of k with some overhead in control circuitry. The hardware is programmable in the sense that the same hardware can be used for any type of tree based codes and it can be easily adapted to implement adaptive codes. The design of a prototype 2-micron VLSI chip based on the algorithm described in this paper for $k = 2$ is presented in a separate paper [MUKH92]. The chip occupies a silicon area of 6.9 x 6.8 square millimeters and it contains 49,695 transistors. The chip has an estimated compression rate of 88 Mbits/sec and a decompression rate of 53 Mbits/sec with a clock rate of 50 MHz. This paper will describe the underlying algorithm and the architecture for this chip and will also present the software algorithms necessary for compilation of the memory map for arbitrary value of k.

The design of the memory architecture for k-bit decode/encode function has lead to the formulation of an open problem, called the **Contiguous Binary Superstring (CBS)** problem. The problem can be informally stated as follows: given a set of m binary strings, $S_1, S_2, \ldots, S_m$, find a superstring S of shortest length such that each $S_i$ is contained in S contiguously and S is the union of $S_i$'s such that no more than one $S_i$ contributes a '1' in any position of S. This problem has the flavor of the multiple string alignment problem [LIMI90, SANK85] and the superstring problem [TARH88], but is quite distinct from them. We have developed a "**greedy**" heuristic algorithm to solve the problem. We will present this algorithm in this paper with an analysis of its complexity. Finding an efficient algorithm with provably good bounds is proposed to be an open problem.

## 2 The Tree Based Codes and the Reverse Tree

By **tree based code**, we mean the set of encodings that can be represented by a binary tree, as shown in Figure 1, as an example. The leaf nodes represent the symbols to be encoded and the sequence of 1's and 0's in the unique path from the root of the tree to the leaf node represent the unique code for that symbol. Tree based codes represent a large class of instantaneously decodable variable-length encoding schemes. For a discussion of these codes and their properties, the reader is referred to the review paper by Lelewen and Hirschberg [LELE87]. For the development of the hardware implementation of the tree based codes, the concept of **reverse binary tree** [MUKH89, MUKH91] will be useful. A reverse binary tree is a labeled binary tree whose leaves and some of the internal nodes represent the symbols to be encoded in the following sense: the sequence of 0's and 1's in the unique path from the node representing the symbol to the root node is the code for the symbol. Given the binary tree representing the encoding scheme, the reverse binary tree can be obtained by the following algorithm:

(i) Obtain the reverse code for each symbol by writing its original code backwards.

(ii) Consider the reverse code for the first symbol and construct a right child to the root node if the first bit is a '1' or a left child if the first bit is a '0'.

(iii) Assuming this newly built node as the parent node, consider the second bit of the reverse code and build a new child node as before. Repeat this step until all the bits of the code for the first symbol are considered.

(iv) Consider now the reverse code for the second symbol. If the first bit is a '0', we need a left child from the root and if the bit is '1', a right child is to be constructed. If the particular child node already exists due to the consideration of a previous symbol, traverse to that node and consider the second bit of the reverse code. The same procedure is applied to all the bits of the code for the second symbol constructing only the missing nodes during each step.

(v) Repeat step (iv) until the reverse codes for all the symbols have been considered.

The resulting tree is the reverse binary tree obtained from the original code. The reverse binary tree for the example tree of Figure 1 is given in Figure 2. The time complexity for the

construction of the reverse binary tree is linearly proportional to the total length of binary codes of all the symbols.

For the purpose of developing multi-bit encoding and decoding schemes, we will define a **k-bit tree** associated with a code as follows: each edge of the tree corresponds to the encoding of a maximum of k bits of the code. If the length of the code is n, it is represented by a sequence of $\lceil n/k \rceil$ labels in the unique path from the root to the leaf of which only the last edge leading to the leaf node could possibly have a label with less than k bits. The tree of Figure 1 is a 1-bit tree; the corresponding 2-bit tree for the same code is shown in Figure 3. In an analogous fashion, we define a **k-bit reverse tree**. In this case, the sequence of symbols read from the leaf to the root of the tree specify the unique code for the symbol. The reverse binary tree of Figure 2 represents a 1-bit reverse tree; the corresponding 2-bit reverse tree is shown in Figure 4. The algorithm to obtain a k-bit reverse tree is obviously very similar to the algorithm for the reverse binary tree as described above and the complexity of construction is linearly proportional to the total length of the binary codes of all the symbols.

# 3 Memory Map of a k-bit Decoding/Encoding Tree

The architecture of **encoder/decoder chip** is based on a memory in which the code trees (both the k-bit decoding tree for decoding the symbol table and k-bit reverse tree for encoding the corresponding symbols) are stored. In this section, we will present a systematic method of mapping the nodes of the tree onto the memory. We will also describe the software to compile the k-bit trees and reverse trees starting from the symbol/code table. We will first discuss the mapping of the decoding tree.

Let there be n nodes in the k-bit decoding tree of which there are p nodes ($p < n$) $N_1, N_2, \ldots, N_p$ which are non-leaf and each having at least two child nodes. The remaining nodes $N_{p+1}, \ldots, N_n$ are either leaf nodes or non-leaf nodes with only one child[4]. Consider one of the nodes $N_i$ ($1 \leq i \leq p$) and assume that it has c child nodes; obviously, $1 \leq c \leq 2^k$. Let the edge leading to the t-th child ($1 \leq t \leq c$), has a label $L_t^i = x_1 x_2 \ldots x_s$ where $s \leq k$ and $x_i$ ($1 \leq i \leq s$) is a binary integer 0 or 1. Define an integer $B_t^i$ associated with $L_t^i$ as

$$B_t^i = \sum_{j=1}^{s} 2^{k-j} x_j$$

[4]In the case of Huffman's k-bit decoding tree, every non-leaf node will have at least two children

The set of numbers $B_1^i, B_2^i, \ldots, B_c^i$ are all distinct since the labels $L_i^i$ obey the prefix property (that is, no label is a prefix of another label). Associate a positive integer variable $M_i$ with node $N_i$ and define a set of c numbers, Mem($N_i$), associated with $N_i$ as

$$Mem(N_i) = \{M_i + B_t^i | t = 1, 2, \ldots, c\}$$

An assignment of integer values to the sets of numbers Mem($N_i$), i=1, ..., p such that no two integer values are equal, will be called a **memory map** of the k-bit decoding tree. Let there be q unassigned nodes constituting a subset of the nodes $(N_{p+1}, \ldots, N_n)$. Map each of these unassigned nodes to a distinct positive integer outside the memory map. Call this set to be a **terminal map** for the k-bit tree. The union of the memory map and the terminal map is called the **total memory map**.

**Example 1:** The 2-bit tree corresponding to a Fibonacci code [LH87, p.276] is shown in Figure 5. The memory map assigns unique positive integers to the children of $N_1, N_2, N_3$, and $N_4$ where

$$\begin{aligned}
\text{Mem}(N_1) &= \{M_1 + 0, M_1 + 1, M_1 + 2, M_1 + 3\} \\
\text{Mem}(N_2) &= \{M_2 + 0, M_2 + 1, M_2 + 3\} \\
\text{Mem}(N_3) &= \{M_3 + 0, M_3 + 2\} \\
\text{Mem}(N_4) &= \{M_4 + 1, M_4 + 3\}
\end{aligned}$$

Assigning $M_1 = 0, M_2 = 4, M_3 = 6, M_4 = 8$ produces a solution as given in Figure 5 by the numbers adjoining each node. For the four remaining unassigned leaf nodes, we can take the terminal map to be $N_5 \rightarrow 10, N_6 \rightarrow 12, N_7 \rightarrow 13, N_8 \rightarrow 14$ producing a total map.

One notes that for the above example, it was possible to map all nodes of the tree (excluding the root node) to a set of consecutive integers. Such a map will be called a **perfect map**. A good map will be the one that maximizes the use of consecutive integers. Assuming the map uses integers 0 through $N-1$ with W unassigned integers, the ratio W/N will be called the **gap** g of the map. A perfect map has no gap. The ratio (n-1)/N will be called the efficiency of the map.

Note, for a perfect map, g=0 and efficiency is 100%. A sufficient condition for a perfect map is known.

**Theorem** : A 1-bit binary decoding tree has a perfect map.

**Proof :** Each non-leaf node of $N_1, N_2, \ldots, N_p$ has two children corresponding to labels 0 and 1. Assigning the first p even integers (viz. 0, 2, ..., 2(p-1)) to the left child of $N_1, N_2, \ldots, N_p$ respectively will lead to a perfect map.

A greedy algorithm (CBS algorithm) for obtaining a memory map for a k-bit decoding tree is presented in Section 4. This algorithm does not produce an optimal memory map (i.e., a map for which W is minimum), but as we will see W can be utilized to map the encoding tree which needs n arbitrarily chosen distinct integers for its memory map. Thus, even a relatively large gap does not lead to any inefficient utilization of the available address space. The problem of obtaining an optimal memory map for the decoder is an open question and will be discussed in Section 4.

The encoding map is created by using the reverse tree. Since each node has only one parent node, the addresses of the nodes can be assigned arbitrarily as long as they are distinct, as shown in Figure 6. But, to simplify the address decoding hardware, we will take the fixed length binary word representing the symbol to be the memory location associated with the symbol.

### Memory Word Format

For k=2, the memory word has the format shown in Figure 7. The fields of the word have different meanings for encoding and decoding operations. For decoding operation, since our objective is to decode 2 bits per cycle, if possible, we need to distinguish between a **regular node**, which is a non-leaf node from which transition to all its children has 2 bits on the edge label (such as the one shown in Figure 8(a)), a non-leaf node with two single bit transitions (Figure 8(b)), non-leaf nodes with a single bit and 2-bit transitions (Figure 8(c) and (d)) and a terminal node (Figure 8(e)). For a regular node, the 2 bit decoding process will proceed to the child node. For (b), (c) and (d), 1 bit transitions lead to terminal symbols and therefore the decoder should output the symbol but backup one bit position to start decoding the next symbol.

The meaning for the t, b and f bits for decoding operation are assigned as follows:

| t | b | f | Type |
|---|---|---|---|
| 0 | 0 | 0 | A regular non-terminal (no backups) |
| 0 | 0 | 1 | A non-terminal with backup on 1 transition |
| 0 | 1 | 0 | A non-terminal with backup on 0 transition |
| 0 | 1 | 1 | A non-terminal with 2 backups on both 1 and 0 transitions |
| 1 | 0 | 0 | A terminal node |

For terminal nodes, the data field (see Fig. 7) corresponds to the value of the decoded symbol; in all other cases, the data field designate a next memory location address.

As we discussed earlier, the whole encoding scheme depends upon the corresponding reverse tree of the symbol table. The integer values assigned to each node of the reverse tree represents address of the memory location where that node is mapped. The content (next address and the encoded bits) of that memory location is simply the integer value assigned to its parent node and the label of the edge leading to its parent node represents corresponding encoded bits.

For 2-bit (k=2) encoding operation, our objective is to encode 2 bits per cycle. The t and b are the encoded bits in a cycle. The f bit is a controlling bit, which indicates the number of bits to be encoded at the last memory cycle. If f=1 at the initial address of the symbol to be encoded, it means that the encoding of the symbol uses odd number of bits and at the last memory memory cycle it outputs the bit t only and bit b is ignored. If f=0, both t and b bits are output bits at the last memory cycle. The data field corresponds to the next address to be fetched from the memory in the next cycle.

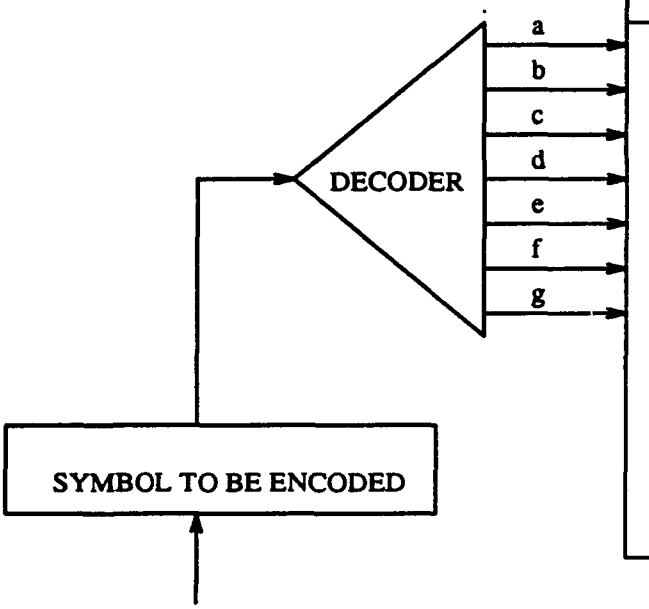
As an example, the memory map of the tree (for the Fibonacci code) of Figure 5 and the memory map of the reverse tree (As shown in Figure 6) are shown in the Tables I and II, respectively. Notice that in Table I, the content of the next address of the memory location of a non-terminal node with a single child is the integer value obtained by subtracting the value of the label leading to the single child node from the address where this child node was assigned. The content of the memory locations corresponding to the leaf nodes is the symbol of code at the leaf node itself.

A bruteforce method of implementing the encoder tree on a memory, will be to store the

| Memory Location | Next Address/Symbol | t | b | f |
|---|---|---|---|---|
| $M_1$ -> 0 | 4 (=$M_2$ ) | 0 | 0 | 0 |
| 1 | 6 (=$M_3$ ) | 0 | 0 | 1 |
| 2 | 8 (=$M_4$ ) | 0 | 0 | 0 |
| 3 | g | 1 | 0 | 0 |
| 4 | 7 (=10-3) | 0 | 0 | 0 |
| 5 | 10 (=12-2) | 0 | 0 | 1 |
| 6 | 11 (=13-2) | 0 | 0 | 1 |
| 7 | e | 1 | 0 | 0 |
| 8 | f | 1 | 0 | 0 |
| 9 | 12 (=14-2) | 0 | 0 | 1 |
| 10 | a | 1 | 0 | 0 |
| 11 | d | 1 | 0 | 0 |
| 12 | h | 1 | 0 | 0 |
| 13 | b | 1 | 0 | 0 |
| 14 | c | 1 | 0 | 0 |

Table I : Memory Map of 2-bit Decode Tree of the Codes below

| Symbol | Code |
|---|---|
| a | 000011 |
| b | 01001 |
| c | 10011 |
| d | 1011 |
| e | 0011 |
| f | 011 |
| g | 11 |
| h | 00011 |




| PRESENT ADDRESS | NEXT ADDRESS | t | b | f |
|---|---|---|---|---|
| 0 | 4 | 0 | 0 | 0 |
| 1 | 8 | 0 | 1 | 1 |
| 2 | 5 | 1 | 0 | 1 |
| 3 | 6 | 1 | 0 | 0 |
| 4 | 6 | 0 | 0 | 0 |
| 5 | 9 | 0 | 1 | 1 |
| 6 | 10 | 1 | 1 | 0 |
| 7 | 5 | 0 | 0 | 1 |
| 8 | 9 | 0 | 0 | 0 |
| 9 | 10 | 1 | 0 | 0 |
| 10 | * | 0 | 0 | 0 |

* encoding of the new symbol

Table II : Memory Map for the Reverse tree (Encoder Table) of Table I.

entire code and the length of the code in each location corresponding to a symbol. This will need a memory size proportional to product of the size of the alphabet and the maximum code length. The codes of length less than the maximum length will be padded with 0's and additional control circuits to extract the correct number of bits using the code length information will be necessary. Our proposed encoding scheme uses the same memory format as the decoding memory with different interpretation of the control bits t, b and f. This enables the encoder/decoder hardware to blend into a combined CODEC (coder/decoder) architecture.

Example 2: We illustrate the decoding process with respect to the symbol 'b' which has a code '01001'. We assume that the memory address register is initialized to 0 (which is the value of $M_0$). For k=2, two bits are decoded in each memory cycle. The decimal equivalent of the first two bits '01' is added to the initial address giving '1' as the first address when the decoding process starts. It is a non-terminal node (t=0) and the next address is 6. The decimal equivalent of the next two bits '00' is added to 6 to give 6 as the next address. The next address of 6 is 11, it is a non-terminal(t=0), the control bits bf='01' indicate that there is a backup on 1 transition. So, only the next single bit (with value 1) is extracted and is appended with a zero bit to give "10", i.e., decimal value of 2. The latter value is added to

decimal 11 to get 13, t=1, indicating it is leaf node and the symbol field 'b' is read out as the decoded symbol.

**Example 3**: We show here how the symbol "b" can be encoded as an example. The initial address for the symbol "b" is 1 as shown in Table II. Now, the content of the memory location 1 shows that the "f" bit is 1 which indicates that the length of the code of "b" is odd. Hence, at the last memory cycle (when next address 10 is reached) only 1 bit will be output. The next address field is 8 and the encoded bits (t and b) are "01". Now the memory location 8 shows that next address field is 9 and the encoded bits are "00". The next address field is 10 and the encoded bits are "10"(t=1, b=0). But since the next address field of memory location 10 is "*", a special symbol designating the last cycle, it outputs t=1 only (ignores b=0) as the encoded bit. Hence, the encoded binary code of the symbol "b" is found to be 01001, which is verified from Table II.

We give below a high-level description of the algorithm to generate the decoding and reverse tree.

**Decoding Tree Algorithm**:

/* Input : the symbol/code table as in Figure 1 and the parameter k. */

/* Output : the k-bit decode tree. */

**begin**

create the root node;

*parent* ← *rootnode*;

**while** (symbol/code table is not empty) **do**

read the binary code of a symbol;

*len* ← *length* of the binary code of the symbol;

**while**(*len* > *k*) **do**

*p* ← *the* decimal value of the next leftmost k-bits of the code;

construct p-th child of the parent if not already constructed;

Associate label of the edge leading to the p-th child with these k-bits;

*parent* ← *p*-th child of parent;

*len* ← *len* − *k*;

**endwhile**;

*p* ← *the* decimal value of the remaining bits of the code;

construct p-th child of the parent if not already constructed;

**endwhile**;

**end**.

Following is the algorithm to generate the k-bit reverse tree from a symbol/code table.

**Reverse Tree Algorithm** :

/* Input : the symbol/code table as in Figure 1 and the parameter k. */

/* Output : the k-bit reverse binary tree. */

**begin**

create the root node;

*parent* ← *rootnode*;

**while** (symbol to be encoded is not exhausted) **do**

$i \leftarrow 0$;

read the binary code;

*len* ← *length* of the binary code of the symbol;

$r \leftarrow$ *len* mod k; /* r is no. of bits to be encoded at the last step */

**if** ( $r > 0$ ) **then**

$p \leftarrow$ *the* decimal value of rightmost r bits of the code;

Construct p-th child of the parent if not already constructed;

Associate label of the edge leading to the p-th child with the rightmost r bits;

*parent* ← $p$-th child of parent;

$i \leftarrow i + r$;

**endif**;

**while** ($i \leq len - 1$) **do**

$p \leftarrow$ *the* decimal value of the next rightmost k-bits;

construct p-th child of the parent if not already constructed;

Associate label of the edge leading to the p-th child with these k-bits;

*parent* ← $p$-th child of parent;

$i \leftarrow i + k$;

**endwhile**;

**endwhile**;

**end**.

# 4 Contiguous Binary Superstring Problem and Compilation of the Memory Map

In the previous section, we described algorithms to construct the k-bit decoding tree and the k-bit reverse tree starting with the symbol-code table. In this section, we present the algorithm that produces the actual memory map.

Given a k-bit decoding tree, the problem is to obtain a total memory map with minimum gap W. A more abstract formulation of the problem can be stated as follows.

For each node $N_i$ $(1 \leq i \leq p)$ (see section 3), associate a $2^k$ bit binary vector $V_i = (a_0 a_1 \ldots a_{2^k-1})$ such that $a_i = 1$ if $(M_i + i)$ is a member of Mem($N_i$); otherwise $a_i = 0$. We say that another binary vector $U = (u_0 u_1 \ldots u_{m-1})$ contains $V_i$ if: (i) $m \geq 2^k - 1$; (ii) there exist $2^k$ consecutive elements in U, $u_{j+0}, u_{j+1}, \ldots, u_{j+2^k-1}$ $(0 \leq j;\ j + 2^k - 1 \leq m)$ such that $u_{j+s} = a_s$ whenever $a_s = 1$ $(0 \leq s \leq 2_k - 1)$. A binary vector $C = (c_0, c_1, \ldots, c_{r-1})$ is said to be a **contiguous binary superstring** (CBS) of a set of vectors $V_1, V_2, \ldots, V_p$ if C contains $V_i (1 \leq i \leq p)$ and C is the bit wise union of $V_i$'s such that if $c_i = 1$ $(0 \leq i \leq r - 1)$ only one of the bits of $V_i$'s aligned to position i of C is 1. The CBS with a minimum number of 0's will be called an **optimal** CBS.

**Example 4**: If $V_1 = 1010, V_2 = 0101$, the string 10100101, 101101, and 1111 all are CBS of which 1111 is the optimal CBS. If $V_1 = 1011$ and $V_2 = 1011$, the optimal CBS is 10111011 with a gap W=2.

It is obvious that a CBS corresponds to a memory map by naturally assigning $C_i$ to the memory location i.

**Example 5**: For our Example 1 with reference to Figure 5, we can write the vectors corresponding to Mem($N_i$), $1 \leq i \leq 4$ as follows $V_1 = (1111)$, $V_2 = (1101)$, $V_3 = (1010)$ and $V_4 = (0101)$. An optimal CBS has the following alignment:

$$V_1 \rightarrow 11111101 \leftarrow V_2$$

$$1010 \leftarrow V_3$$

$$0101 \leftarrow V_4$$

$$CBS = (111111111101)$$

The gap W=1 at location 10 which can now be assigned to the leaf node $N_5$ results in a perfect total map.

The CBS problem has the flavor of the multiple alignment problem [SANK85] and the problem of determining the shortest superstring; but it is different and may still be NP-hard. We will present a "**greedy**" heuristic algorithm below. Note, for the design of combined decoder/encoder memory map, obtaining the optimal CBS is not that crucial because the gap locations can be used up by the encoder map which needs approximately 50% unassigned but freely bound memory addresses.

We need a few definitions. If two vectors $V_i$ and $V_j$ are complimentary, they will be said to align with **0-slide** to form a CBS. In general, if $V_i$ needs a relative shift of s ($0 \leq s \leq 2^k$) with respect to $V_j$ for obtaining a CBS of $V_i$ and $V_j$, it will be called an **alignment of s-slide**. In the above example, the pair $V_1$ and $V_2$ can align with 4-slide, and $V_3$ and $V_4$ align with a 2-slide. The greedy algorithm can be described as follows: given the set of vectors $S = (V_1, \ldots, V_p)$, obtain the pairs of 0-slide vectors. Delete these pairs and add the corresponding CBS's to S. Choose a pair of vectors in S that have a 1-slide alignment. Delete the pair and add the corresponding CBS in S. Keep repeating the step until no more 1-slide alignment can be found. Then, successively repeat 2-slide and 3-slide alignments. When no further alignment is possible, concatenate the vectors in S to obtain the single CBS for the original set of vectors. Formally, the algorithm for software implementation is presented below :

**Greedy Algorithm for the CBS problem**

Let $S = \{V_1, V_2, \ldots, V_p\}$ be the given set of vectors. The greedy algorithm to find CBS of S has the folowing steps:

**Begin**

**Repeat**

*max* ← *length* of the longest vector in S;

**for** j=0 to max-1 **do**

**begin**

Find distinct pair of vectors $V_i$ and $V_j$ in S which
form alignment of j-slide to form a CBS V;

Delete $V_i$ and $V_j$ from set S and add V into S;

**end**

**until** no more allignment possible;

*CBS* ← *Concatenation* of all the vectors in S;

**return** with **CBS**;

**End**.

**Analysis of the time complexity of the CBS algorithm** :

To test whether each pair of binary vectors ($V_i$, $V_j$) form CBS, it needs O(p) comparisons steps. In each pass there might be maximum p number of such test for CBS. Hence, each pass needs $O(p^2)$ comparison steps to test for alignment of s-slide to form a CBS (for any value of s). To compute the final CBS, O(p) such passes are required in our heuristic algorithm. Hence, time complexity of the above heuristic algorithm is $O(p^3)$ with p number of vectors in the initial set.

One should be noted that the CBS formed over a set of binary vectors is not necessarily unique. The same algorithm may generate different CBS depending upon the order of comparison of the vectors in S. A C program for the above algorithm has been implemented and used for the memory map of 2-bit compression/decompression with different types of variable-length codes and the JPEG baseline Huffman table for AC and DC coefficients of luminance and chrominance codes. We compared the results with both 1-bit decoding/encoding and 2-bit decoding/encoding scheme. For 1-bit decoding, the memory map for the luminance AC coefficient code table has a size of 645 memory words. Applying the CBS algorithm for memory map, the same 1-bit decoding table needs 480 words. For 2-bit decoding, the memory map obtained by the CBS algorithm for the same table needs only 226 memory words. To store the code table for chrominance AC coefficients, total number of memory words required for the 1-bit decoding scheme (without applying the CBS algorithm) is 643, whereas the same shceme when the CBS algorithm is applied needs only 478 memory words. The code table for 2-bit decoding scheme using the CBS algorithm needs 223 words, i.e. less than half of the number of memory words required for 1-bit decoding scheme. The memory map for the encoder tree (reverse tree) needs 747 words for the luminance AC coefficient code table and 724 words for the chrominance AC coefficient code table in the 2-bit decoding/encoding scheme. The number of codes in each table (both for AC and DC chrominance and luminance codes of JPEG baseline) is 162 and most of the

codes are 16 bits long. In most of the 2-bit tree examples that we have experimented, we observed that W is less than 20%. In the worst case, a trivial algorithm (concatenation of all the vectors $V_1, \ldots, V_p$ each having a single 1, which is a rare case) gives 75% gap which is the upper bound. If each vector $V_i$ looks like 1011...11, the optimal CBS will have 1/2k wastage (for k=2, it is 25%). Thus the greedy algorithm works well from practical point of view, but still obtaining a provably good heuristic is an open challenge.

# 5 The 2-bit Decoder/Encoder Architecture

**Decoding Algorithm** :

The essential hardware to execute the decoding algorithm consists of a memory (MM), where MM[x] denotes the content of memory in memory location x, a memory address register (MAR) that holds the address for a memory access, a memory data register (MDR) which contains the accessed memory word and a two bit register A[1,0] where the edge label for the next edge to be traversed is assembled during the decoding process. It is assumed that the decoding tree has been compiled ahead of time and initially the MAR contains address of the beginning of the memory table which is the address of the root node of the decoding tree. To be specific, assume MDR has 12 bits, denoted MDR[11, ..., 0] and MAR has 9 bits MAR[8, ..., 0].

**Begin**

**while** (bit string to be decoded not exhausted) **do**

$MAR \leftarrow RootNodeAddress$;

$MDR \leftarrow$ *initial* value corresponding to the root node;

/* This initial value is supplied by the preprocessor */

/* After every memory fetch MDR[11, ..., 3] contains */

/* the " NEXT ADDRESS/SYMBOL " field of the memory word */

/* and MDR[2,...,0] contains t,b,f bits of the word. */

**while** (t=0) **do**

**begin**

$A[1] \leftarrow$ next bit on input stream;

**case**

(f=0, b=0) : $A[0] \leftarrow$ next bit on input stream;

(f=1, b=1) : $A[0] \leftarrow 0$;

(f=1, b=0) : $A[0] \leftarrow 0$;

(f=0, b=1) : $A[0] \leftarrow$ next bit on input stream;

**endcase**

$MAR \leftarrow MDR[11,\ldots,3] + A$;

$MDR \leftarrow MM[MAR]$;

**end**

**endwhile**

Output the decoded symbol;

**endwhile**

**end**.

**Encoding Algorithm** :

The least significant bit (f) of the memory word is called the parity bit which actually indicates whether the code of the symbol is of odd length or it is of even length (if f=1 then at the last step only 1 bit will be emitted). The next least significant 2 bits (i.e., t and b bits) are the encoded bits corresponding to a symbol. The remaining bits of the word designate the next address of the memory location to be accessed.

**Begin**

Load the encoder table;

$MAR \leftarrow$ *beginning* address of the symbol to be encoded;

$MDR \leftarrow MM$[MAR];

/* After every memory fetch MDR[11,...,3] contains the "NEXT ADDRESS" field */

/* of the memory word fetched and MDR[2,...,0] contains t,b,f bits respectively. */

$MAR \leftarrow MDR[11\ldots3]$;

$F \leftarrow f$;

**while** ($MAR \neq$ *a* special address ("*")) **do**

**begin**

```
            output the encoded bits (t,b);
            MDR ← MM[MAR];
            MAR ← MDR[11...3];
        endwhile;
        if(F = 1)
         then
            output the "t" bit only; /* MDR[2] */
         else
            output both "t" and "b" bits; /* MDR[2], MDR[1] */
        endif;
end.
```

The decoder and the encoder can be combined into a single VLSI chip architecture as shown in Figure 9. The decoder 2-bit tree and its reverse tree are preloaded into the memory. If there is any gap in the decoder memory map, this memory space can be utilized by the encoder memory map since many of its non-leaf nodes can be freely placed anywhere in the memory as we discussed earlier. The beginning addresses of these tables are made available to the global control. When the D/E(decode/encode) signal is set to 1, the machine works as a decoder; if it is set to 0, it works as an encoder. The decoder operation proceeds as follows. The decoder control generates shift signal to read one or two bits from the input bit string(depending upon the values of t, b, f bits) which is assembled into a number C that is added to the next address in the Adder circuit. The demultiplexor **DMUX2** selects t and b bits to the control which is able to generate all local control signals. If a terminal symbol is reached, the demultiplexor **DMUX1** puts the content of MDR (excluding the three least significant bits) to the output buffer **SYMBOL**. In essence, the hardware performs the decoding algorithm as presented at the beginning of this Section. For encoding operation, the input symbols are used to access the memory via Address Decoder, the second and third least significant bits of the memory data register (MDR) are selected for output to the first-in-first-out register (FIFO). The control flip-flop F, set by the length code detector reads only one or two bits into the FIFO depending on the length of the label in the reverse tree (see discussion earlier). Note, during the encoding operation, the adder circuit could be bypassed since the next address is directly read from MDR. During decoding, the address computation and the memory access could be easily pipelined for successive pairs of bits to be decoded resulting in higher throughput.

The hardware can easily be reconfigured to do single bit decoding/encoding operation. In this case, we will use the 1-bit tree and the reverse binary tree. We can avoid the addition cycle for next address computation in the decoder by shifting the next address left one bit and by simply appending [BHED90] rather than adding, the terminal bit t (the backup bit is not required). Of course, the 'next address' has to be half of the original address, that is, the address as derived in the proof of Throrem 1. The control circuitry can be much more simplified, since both encoding and decoding processes handle one bit in every cycle.

The hardware described above is programmable in the sense that any tree based code (Huffman, Shannon-Fano, Elias, etc.) can be implemented on the same hardware. The preprocessing step consists of preloading the memory with the appropriate memory maps. In fact, memory map for several codes can simultaneously exist on the memory and switching from one code to the other simply amounts to making the beginning address of the maps available to the control. The architecture is therefore easily adaptable to adaptive codes. This can be done by implementing the memory as a two-port memory. The write port of the memory can be used to load to a different part in memory an updated memory map computed by the host processor based on the most recent statistics of the frequency of distribution of symbols. At appropriate intervals of time, the status of the read and write ports can be switched, thus adapting to the new codes.

The architecture described above has been simulated (using C programming language) and the results obtained from the simulated runs indicate that the 2-bit decode/encode hardware approximately doubles the throughput of compression/decompression and uses almost half the amount of memory compared to the 1-bit decode/encode scheme. There is, however, some overhead in the form of additional hardware viz. the flip-flops, shift registers, the adder and the control circuits. A question that naturally arises is: what is the value of k for which a k-bit decoder/encoder represents the best tradeoff between hardware complexity and throughput. The following discussion points out some general features.

For an arbitrary k, we can say on the average that the height of the decoder tree will be reduced by a factor of $1/k$ and the size of the memory map will be decreased by a factor $1/2^k$ with an increase in word size by $log_2 k$ additional bits. A speedup of k in the decoding/encoding process compared to the case when k=1 will occur in most situations. We need however $s = log_2 k$ backup bits $b_1, b_2, \ldots, b_s$ to indicate the possibility of a potential backup with 0, 1, 2, ..., k-1 bits in the decoder and same number of control bits to indicate how many of the encoded bits represent valid output bits. The reading of the input bits to

the input buffer also has to be handled by a shifter which can shift 1, 2, 3, ..., k bits etc. Thus, even if we assume that the cost of control circuits is linearly proportional to k, we can achieve a linear speedup in throughput with a factor of $2^k$ in saving memory space.

### 5.1 VLSI Chip Implementation

The chip was implemented in 2-micron SCMOS p-well technology using a standard-cell and micro design approach. The design uses a 2-phase non-overlapping clocking scheme. The chip has been fabricated by MOSIS. The registers, multiplexers, and logic gates were designed as standard cells and the 512x12 static RAM was implemented as a full custom macro. The Cadence design tools running on SUN workstation were used for the entire design. The design approach was to design the standard cells and the RAM and perform automatic placement and routing. The chip occupies a silicon area of 6.9 x 6.8 $mm^2$ and contains 49,695 transistors. There are 55 pins on the chip or I/O and power connections. The chip has capability of an estimated compression rate of 88 Mbits/sec and a decompression rate of 53 Mbits/sec with a clock rate of 50 MHz. The detail design of the chip is presented in a separate paper[**MUKH92**].

## 6 Conclusion

We have presented a memory based architecture for the design of special purpose hardware for real-time compression/decompression of data. A VLSI chip implementing a 2-bit encoding/decoding (CODEC) architecture has been built and tested. The simulation of the JPEG baseline compression/decompression scheme has produced improvements in both size of the memory and the speed of compression/decompression algorithm. Since the architecture is memory based, it is expected that commercial chips based on the basic idea of multi-bit decoding/encoding will be a viable cost-effective approach for building special-purpose CODEC systems. A key feature of the architecture is that it is programmable in the sense that switching from one code to other simply means reloading the memory with new tables. If the reloading is done from time to time, the chip would be capable of supporting adaptive codes.

## 7 Acknowledgement

This work is partially supported by a grant from PMTRADE Contract No. N61339-92-C-0016 from DOD/NTSC.

## References

[ABRA63] Abramson, N. 1963. Information Theory and Coding. McGraw-Hill, New York.

[ARPS88] Arps, R. B., Truong, T. K., Lu, D. J., Pasco, R. C., Friedman, T. D., "A multi-purpose VLSI chip for adaptive data compression of bilevel images", IBM Journal of Research and Development, Vol. 32, No. 6, November 1988.

[BASS85] Bassiouni, M., "Data compression in scientific and statistical data bases", IEEE Transactions on Software Engineering, Vol. SE-11, NO. 10, October 1985, pp. 1047-1058.

[BHED90] Bheda, H. and Mukherjee, A.,"High Performance Still Picture Codec Architecture", (manuscript pending submission).

[BROM91] Bromley,D. Alan,"Grand Challenges: High Performance Computing and Communications" Office of Sciene and Technology Policy, Washington D.C. ( Supplement to the President's Fiscal Year 1992 Budget)

[BUNT90] Bunton,S. and Borriello,G. "Practical Dictionary Management for Hardware Data Compression," Sixth MIT Conference on Advance Research in VLSI, Cambridge,Mass.,1990.

[CLAR85] Clarke r. J., "Transform Coding of Images", Academic Press, 1985.

[CCUB90] C-Cube Microsystems, "CL550A JPEG Image Compression Processor", Preliminary Data Book, Feb. 1990.

[ELIA75] Elias, P., "Universal Codeword Sets and Representations of the Integers", IEEE Trans. Inf.Theory 21, 2(Mar.1975), 194-203./

[FANO49] Fano, R. M., "Transmission of Information", M.I.T. Press, Cambridge, Mass., 1949.

[FRAE85] Fraenkel, A. S., and Klein, S. T., "Robust Universal Complete Codes as Alternatives to Huffman Codes", Tech. Rep. CS85-16, Dept. of Applied Mathematics, The Weizmann Institute of Science, Rehovot, Israel, 1985.

[GALL78] Gallager, R., "Variations on a theme by Huffman", IEEE Trans. on Info. Theory, Vol. IT-24, NO. 6, 1978, pp. 668-674.

[GEY83] F. Gey et. al, "Compute Independent Data Compression for Large Statistical Data Bases", Proc. 2nd Intl. Workshop Statistical Database Management, 1983.

[GONZ85] Gonzalez-Smith, M. and Storer, J., "Parallel Algorithms for data compression", JACMD, Vol. 19, 1986, pp.11-19.

[GREE88] Green, J. L., "Space Data Management at the NSSDC: Applications for Data Compression", Proc. of the Scientific Data Compression Workshop, Snowbird, Utah, 1988.

[HAWT82] Hawthorn, P., "Microprocessor assisted tuple access decompression and assembly for statistical data base systems", Proc. VLDB, 1982, pp. 223-233.

[HAZB82] Hazboun, K. and Bassiouni, M., "A multi-group technique for data compression", Proc. ACM SIGMOD Int. Conf. on Management of Data, 1982, pp. 284-292.

[HUFF52] Huffman, D., "A method for the construction of minimum redundancy codes", Proc. IRE, Vol. 40, 1952, pp. 1098-1101.

[LEA78] Lea, R., "Text compression with an associative parallel processors", The Computer Journal, Vol. 21, No. 1, 1978, pp. 45-56.

[LELE87] Lelewer, D. A., and Hirschberg, D. S., "Data Compression" ACM Computing Survery, Vol. 19. NO. 3, September 1987.

[LIMI90] Li, Ming, "Towards a DNA Sequencing Theory", Proc. Foundation of Computer Science, St. Louis, 1990.

[MILL88] Miller, R. B. et. al, "Science Data Management: Subpanel Report", Proc. of the Scientific Data Compression Workshop, Snowbird, Utah,1988.

[MUKH89] Mukherjee, A., "Huffman code translator," U.S. Patent, Serial No. 038,039, April 1989.

[MUKH90] Mukherjee, A., Ranganathan N., and Bassiouni, M. A., "Adaptive and Pipelined VLSI Designs for Tree-Based Codes". IEEE Trans. on Circuits and Systems, March, 1991.

[MUKH91] Mukherjee, A., Behda, H., Bassiouni, M. A., Acharya, T., "Multibit Decoding/Encoding of Binary Codes Using Memory Based Architecture", Proc. Data Compression Conference, Snowbird, Utah, April,1991.

[MUKH92] Mukherjee, A, Flieder, J and Ranganathan, N, "MARVLE : A VLSI chip of a Membry Based Architecture for Variable Length Encoding & Decoding", Proc. Data Compression Conference, Snowbird, Utah, March 24-28, 1992.

[NETR88] Netravali, A.N., and Haskell, B.G., "Digital Pictures," Plenum Press, 1988.

[RICE90] Rice, R. F., Yeh, P., Meils, W.,"Algorithm for a very high speed universal noiseless coding module", Jet Propulsion Lboratory, Pub. No.91-1, Pasadena, Ca.

[SANK85] Sankoff, D., "Simultaneous Solution of the RNA Folding, Alignment and Photosequence Problems", SIAM J. of Applied Mathematics, Vol. 45, No. 5, 1985.

[SHAN49] Shannon, C. E., and Weaver, W., The Machematical Theory of Communication. University of Illinois Press, Urbana, III.1949.

[STOR82] Storer, J. and Szymanski, T. "Data compression via textual substitution", JACM, Vol. 29, No. 4, 1982, pp.928-951.

[STOR85] M. E. Gonzalez Smith and J. A. Storer, "Parallel algorithms for data compression", Journal of the ACM, 32(2):334-373, April 1985.

[STOR88] Storer, J. "Data Compression Methods and Theory", Computer Science Press, 1988.

[STOR90] Storer, J. A., Reif, J., H. and Markas, T.,"A massively parallel VLSI design for data compression", Proc. of the IEEE Workshop on VLSI Signal Processing, 1990.

[SUN89] Sun, M.T., Chen, T.C. and Gottlieb, A.M., "VLSI Implementation of a 16x16 Discrete Cosine Transform", IEEE Trans. on Circuits and Systems, Vol. 36(4), pp. 610-617, April, 1989.

[TARH88] J. Tarhio and e. Ukkonen,"A greedy Approximation algorithm for Constructing Shortest Common Superstrings", Theoretical Computer Science, Vol. 53, 1988.

[VETT86] Vetterli, M., and Ligtenberg, A., "A Discrete Fourier-Cosine Transform Chip", IEEE Journal on Selected Areas in Communications, Vol. SAC-4(1), Jan. 1986.

[VENB91] Venbrux, J., Liu, N., Liu, K., Vincent, P., Merrel, R., "A very High speed lossless compression/decompression chip set", Jet Propulsion Laboratory, Pub. No. 9130, Pasadina, Ca.

[WALL90] Wallace, G., Pennebaker, W.B. and Mitchell, J. L., Draft Proposal for the Joint Photographic Expert Group(JPEG), JPEG-8-R6, June 24, 1990.

[WELC84] Welch, T., "A Technique for High-Performance Data Compression", Vol. 17, No. 6, 1984.

[WITT87] Witten, I.H., Neal, R.M., and Cleary, J.G., "Arithmetic Coding for Data Compression", Commun. ACM 30, 6 (June), 520-540.

[WOLF90] Ronald Zito-Wolf. "A Broadcast/Reduce Architecture for High-Speed Data Compression",Proc. IEEE workshop on Data Compression, 1990.

[ZIVL78] Ziv, J., and Lempel, A., "Compression of Individual Sequences Via Variable-rate Coding", IEEE Trans. Inf. Theory 24, 5 (Sept. 1978), 530-536.

| Symbol | Code |
|---|---|
| a | 010 |
| b | 011 |
| c | 100 |
| d | 00 |
| e | 101 |
| f | 110 |
| g | 1110 |

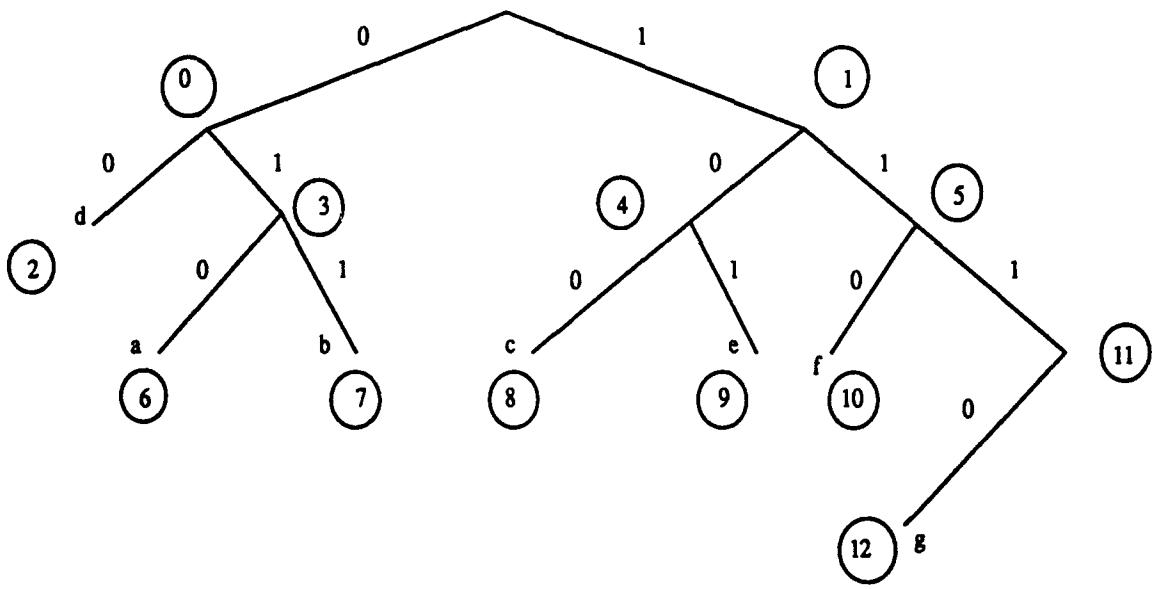



Figure 1: Tree Representing the above Variable Length Codes

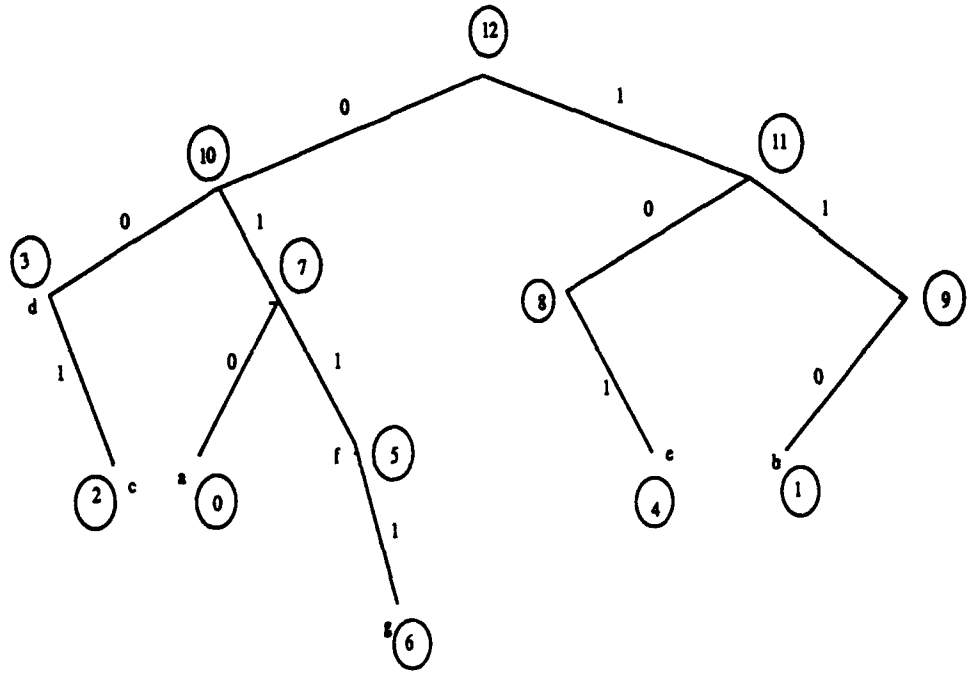



Figure 2: The 1-bit reverse tree corresponding to the 1-bit tree of Figure 1

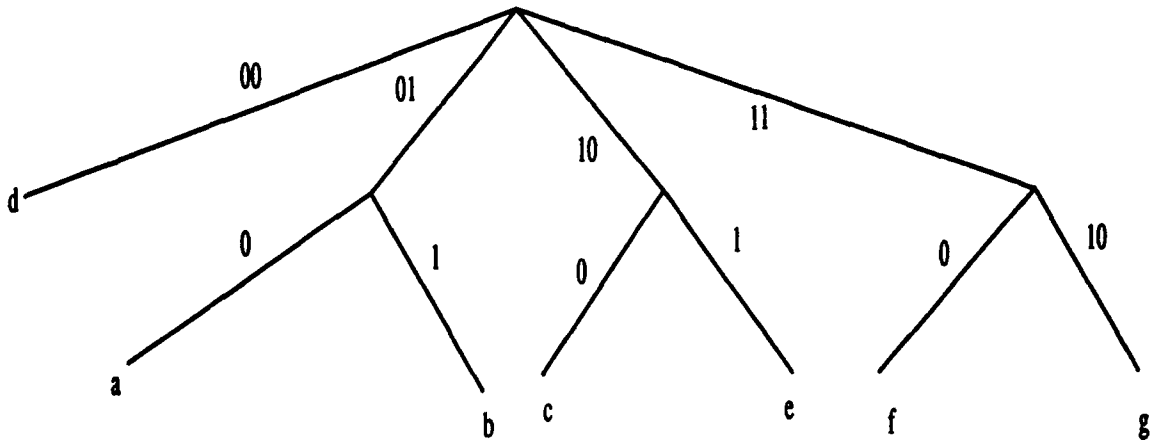



Figure 3: The 2-bit tree corresponding to the 1-bit tree of Figure 1

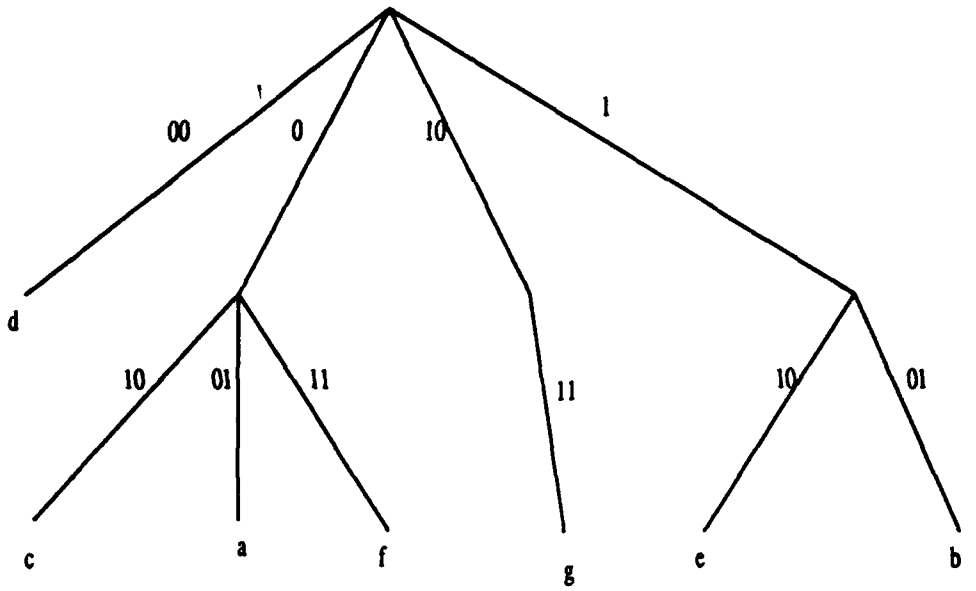



Figure 4: The 2-bit reverse tree corresponding to the 1-bit reverse tree of Figure 2

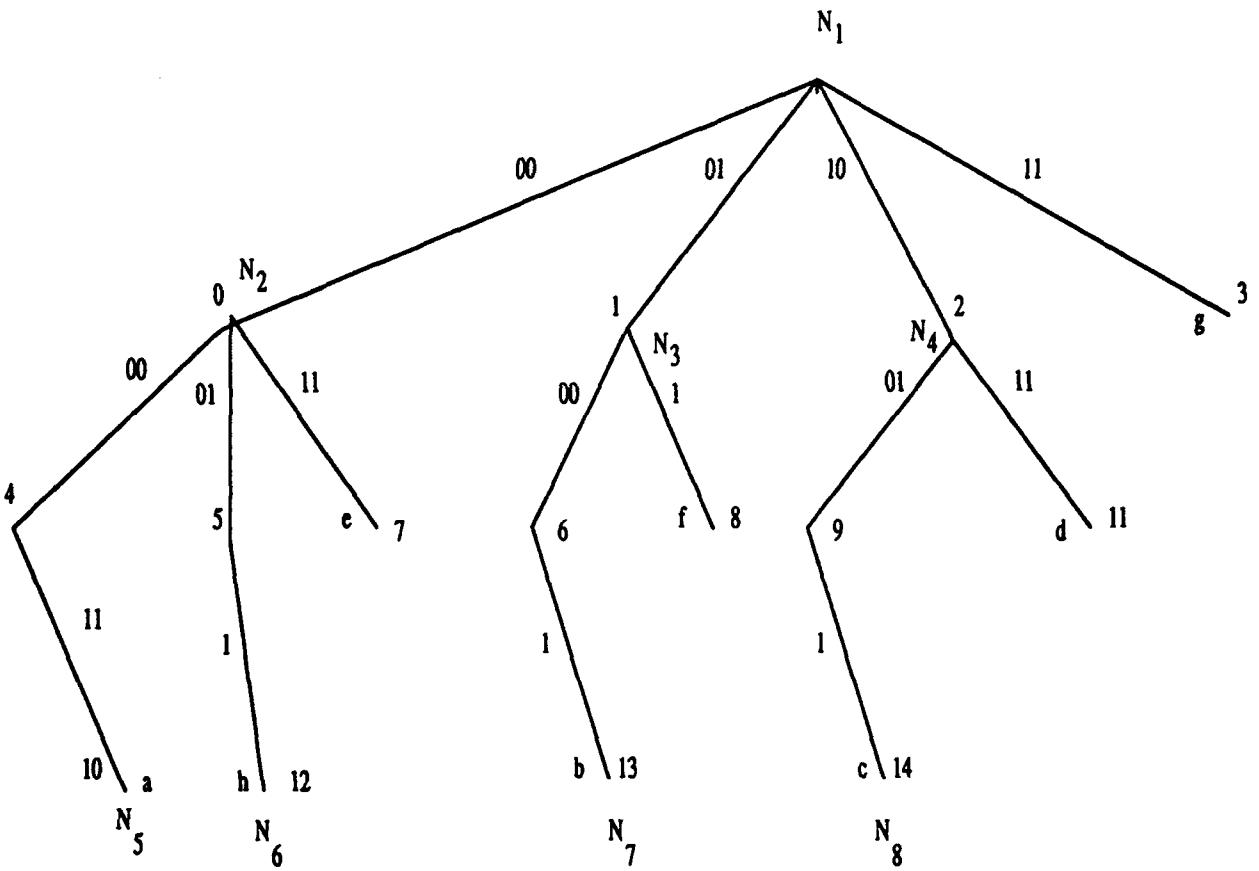



Figure 5: A 2-bit tree for a Fibonacci code

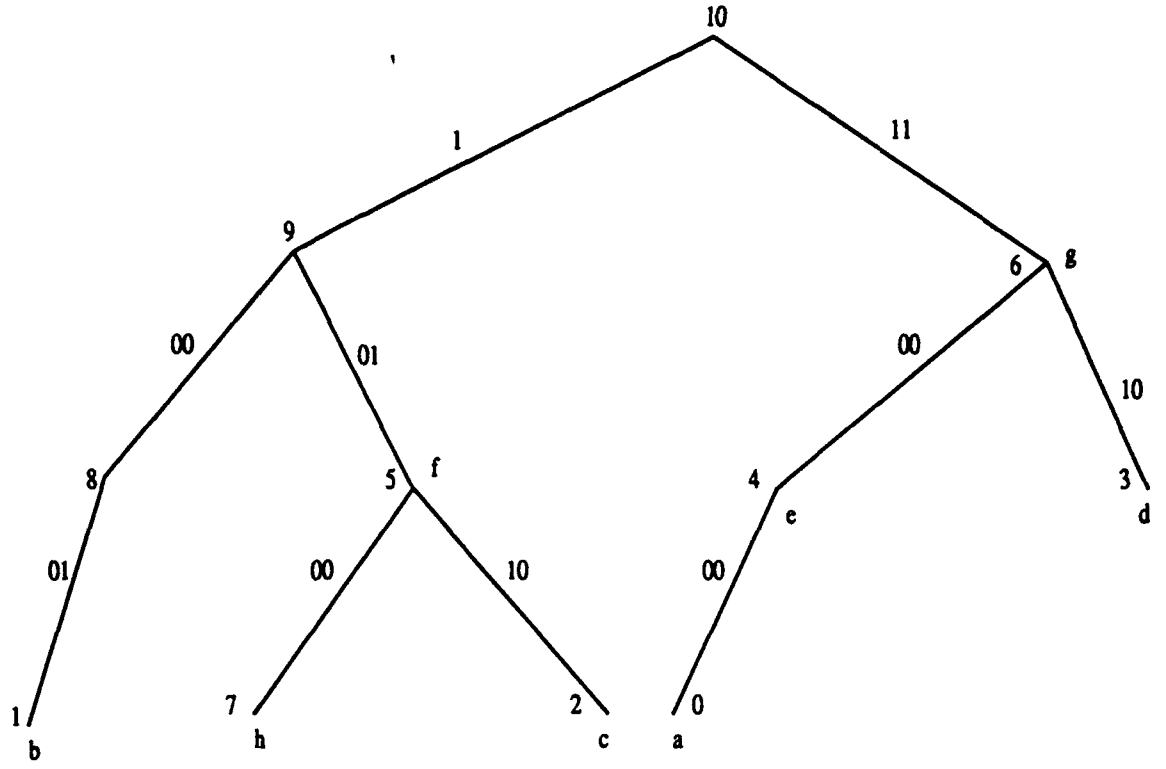



Figure 6: The reverse tree for the tree of Figure 5

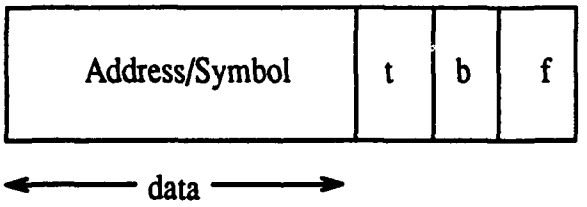



Figure 7: Format of the memory word

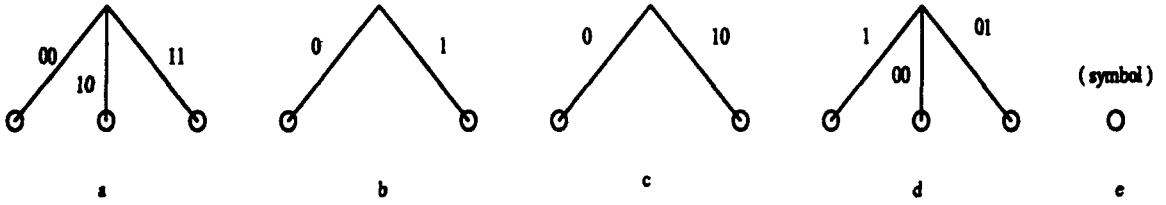



Figure 8: Possible labels at the lowermost level

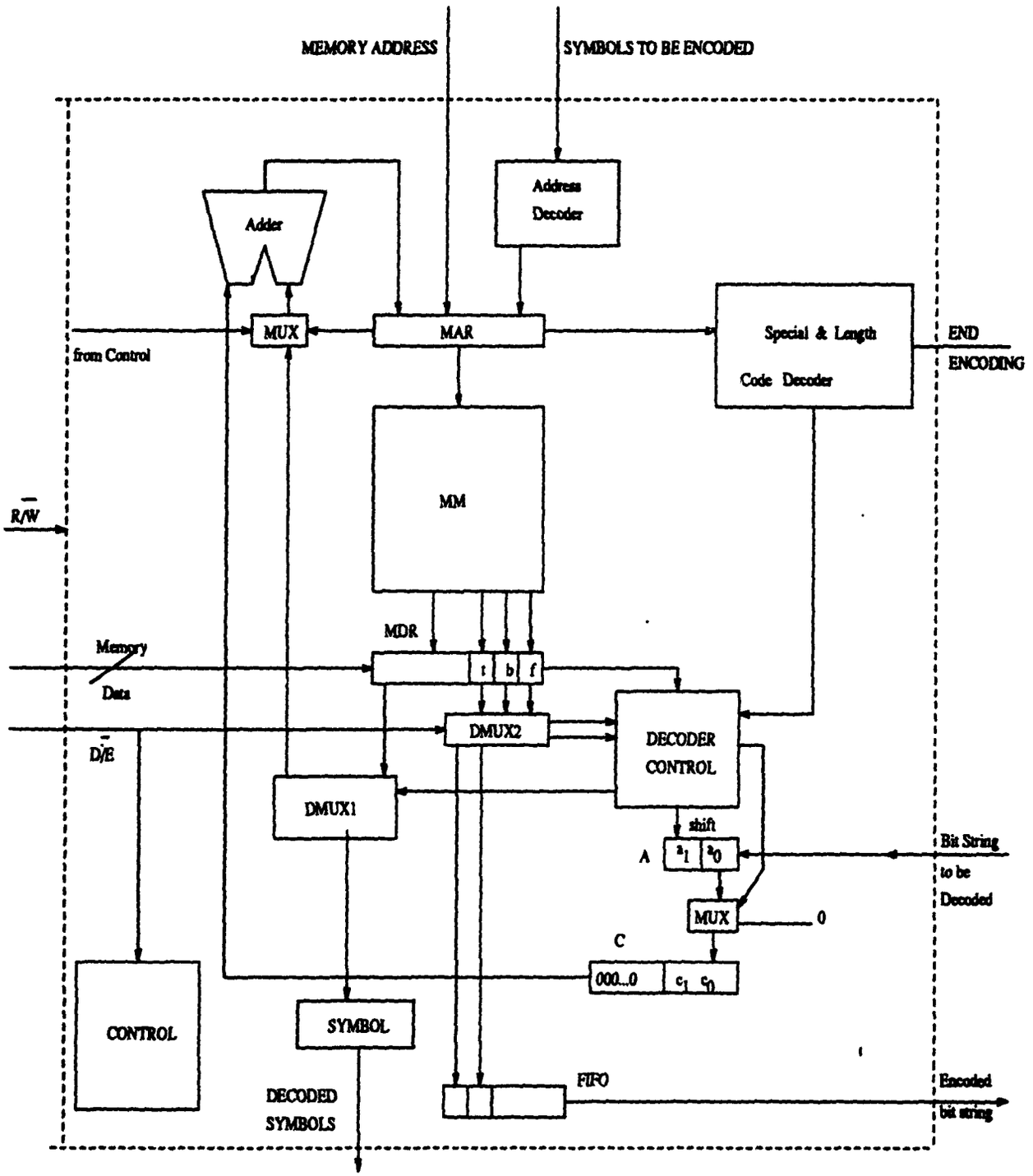



Figure 9: Decoder/Encoder Architecture